

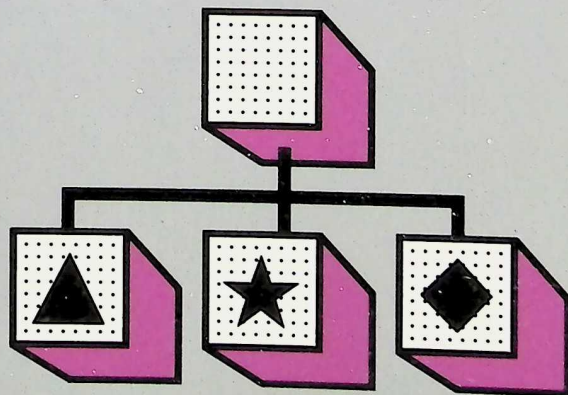
TURBO PASCAL

Version 2.00
S/N: 353154

Async Professional 2.0TM

NOW
Compatible with
Borland Pascal
7.0

A powerful communications toolkit for DOS applications. Includes ■ OOP and procedural interfaces ■ UART, Fossil, and DigiBoard device support ■ Async debugging tools ■ ZMODEM, YMODEM, XMODEM, CIS B+, and Kermit protocols ■ Class I, Class II, and CAS fax ■ ZIP and LZH data compression, and more. Full source and no royalties.



TURBO 
Power

Async Professional

User's Manual

Copyright (c) 1991, 1993, TurboPower Software
All Rights Reserved

Second Edition November 1993

Order line (US and Canada): 800-333-4160
Elsewhere: 719-260-9136

Technical Support
TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009

Telephone Support: 719-260-6641
9am to 5pm Mountain Time, Mon-Fri
Fax: 719-260-7151
BBS: 719-260-9726

CompuServe: 76004,2611
PCVENB Section 6

Async Professional is copyright (c) 1991, 1993 by TurboPower Software, all rights reserved.
Portions are copyright (c) 1987-1989 by Information Technology Ltd., and are used under license
to TurboPower Software. All TurboPower Software products are trademarks or registered
trademarks of TurboPower Software. Other brand and product names are trademarks or registered
trademarks of their respective holders.

Table of Contents

1. Introduction	1
A. Requirements.....	2
B. Installation	3
C. Configuration.....	10
D. Organization of this Manual	15
E. Diagnostic Programs.....	17
1. UART Identification Program: UARTID.....	17
2. Communications Test: COMTEST/COMTESTO.....	18
F. Demonstration Programs	22
1. A Simple Communications Program: SIMPCOM.....	22
2. A Sophisticated Communications Program: OOPCOM.....	25
3. Installation Program for OOPCOM: OOPINST	37
4. A Sophisticated Fax Program: TPFAX.....	43
G. The Help System.....	52
H. Calling for Technical Support.....	59
I. Purchase Agreement.....	61
2. Principles of Operation	63
A. Overview	63
B. Basics of Asynchronous Serial Communication.....	64
C. Async Professional Architecture.....	76
D. Object Hierarchy	80
E. Using Async Professional in Protected Mode DOS	82
F. Protected Mode DLLs.....	84
3. Timer Functions	87
A. Overview	87
4. The Abstract and Device Layers	95
A. Overview	95
B. The Abstract Layer: APPORT	96
C. UART Device Layer: APUART	120
D. Fossil Device Layer: APFOSSIL.....	133
E. DigiBoard Device Layer: APDIGI14	137
F. BIOS Interrupt \$14 Device Layer: APINT14	140
G. Core Routines.....	142
H. Miscellaneous Routines: APMISC	159
I. Error Handling.....	166
5. The Interface Layer	173
A. Overview	173
B. OOP vs. Non-OOP Interface	174
C. The Interface Layer: APCOM/OOCOM.....	178

6. Modems.....	239
A. Overview.....	239
B. OOP vs. Non-OOP Interface	240
C. Modem Control: APMODEM/OOMODEM.....	242
D. U.S. Robotics Courier Modem: APCOUR	293
E. Microcom Modem: APMCOM	294
7. File Transfer Protocols	295
A. Overview.....	295
B. OOP vs. Non-OOP Protocols	296
C. Abstract Protocol Functions: APABSPCL/OOABSPCL	298
D. Xmodem Protocol: APXMODEM/OOXMODEM	349
E. Ymodem Protocol: APYMODEM/OOYMODEM	361
F. Zmodem Protocol: APZMODEM/OOZMODEM.....	368
G. Kermit Protocol: APKERMIT/OOKERMIT	384
H. CompuServe B+ Protocol: APBPLUS/OOBPLUS.....	406
I. Simple Text File Transfer: APASCII/OOASCII	426
J. Demonstration Programs	436
1. File Transfer Program: FX/FXO.....	436
2. Background File Transfer Program: BACK/BACKO	438
8. Text File Device Drivers.....	441
A. Overview.....	441
9. Terminal Emulation	447
A. Overview.....	447
B. OOP vs. Non-OOP Emulation.....	448
C. Procedural Terminal Emulation: APANSI	450
D. Object-Oriented Terminal Emulation: OOEMU	456
E. Terminal Window Object: TERMWIN.....	466
F. Demonstration Programs	494
1. Terminal Window Test: TERMTEST	494
10. Fax Support.....	497
A. Overview.....	497
B. OOP vs. non-OOP Interface	500
C. Document Conversion: APFAXCVT/OOFAXCVT.....	503
D. Abstract Fax Send/Receive: APABSFAX/OOABSFAX	530
E. Class 1/2 Fax Send/Receive: APFAX12/OOFAX12.....	559
F. CAS Fax Send/Receive: APFAXCAS/OOFAXCAS	589
G. Demonstration Programs	618
1. Simple Send Fax Program: SIMPSND/SIMPSNDO.....	618
2. Simple Receive Fax Program: SIMPRCV/SIMPRCVO	620
3. Convert to Fax: CVT2FAX/CVT2FAXO	621
4. Fax File Display: SHOWFAX/SHOWFAXO.....	622
5. Fax File Print: PRNFAX/PRNFAXO	624
6. Convert Fax to PCX: FAX2PCX/FAX2PCXO	625
7. Background TSR Fax Receiver: RFAX/RFAXO	626
8. Dedicated Fax Receiver/Printer: FAXSRVR/FAXSRVRO.....	628

11. Data Compression	631
A. Overview	631
B. Archive Utility Functions: APARCHIV/OOARCHIV	632
C. LZH Archives: APLZH/OOLZH	644
D. ZIP Archives: APZIP/OOZIP	681
E. Demonstration Programs	719
1. LZH Archive File Compress: LZH/LZHO	719
2. LZH Archive Directory View: LZHV/LZHVO	721
3. LZH Archive File Extract: LZHX/LZHXO	723
4. ZIP Archive File Compress: ZIP/ZIPO	725
5. ZIP Archive Directory View: ZIPV/ZIPVO	727
6. ZIP Archive File Extract: ZIPX/ZIPXO	730
12. Appendices	733
A. Glossary of Communications Terms	733
B. Error Codes	739
C. Debugging Communications Programs	743
D. Serial Port Diagram	747
E. What's New in Version 2.0	748
Identifier Index	749
Index	759

1. Introduction

Asynchronous communication has been part of many Turbo Pascal applications for years. Although there are a variety of public domain and shareware routines to handle various facets of asynchronous communications, there has never been a comprehensive library of quality routines that addressed all phases of the communications problem. That's where Async Professional comes in.

Async Professional offers a set of modern, high-quality routines to manage your communications today and, perhaps equally important, tomorrow as well. That is, one of our primary objectives was to make sure that programs based on Async Professional can easily be adapted to upcoming technologies and standards in serial- and modem-based communications.

Additionally, Async Professional takes advantage of object-oriented programming to offer a sensible easy-to-use library of communication objects. Such objects will help you build applications faster and make them easier to maintain.

For those programmers who prefer to use procedural programming techniques, Async Professional also offers all of its features and power in a traditional, procedural interface.

Async Professional provides many features that set it apart from its competitors:

- interrupt-driven serial I/O up to 115K baud
- support for UART, Fossil, Digi14, and other device drivers
- support for 16550 buffered UARTs
- industry standard protocols (including Zmodem, Kermit, and B+)
- fax support (send and receive)
- fax file conversion (text, PCX, and TIFF)
- fax printing and viewing utilities
- file compression and decompression (ZIP and LZH)
- written in Pascal and assembly language, with full source provided
- built-in debugging tools
- OOP and procedural interfaces
- modem support
- terminal emulation support
- text file device drivers

Async Professional is for use in a DOS environment (Windows environment is not supported).

A. Requirements

To use this package, you must have the following:

1. An IBM PC, XT, AT (or close compatible), or PS/2 running DOS 3.30 or later.
2. Turbo Pascal version 6.0 or 7.0 or Borland Pascal 7.0.
3. At least 512KB of RAM.
4. A hard disk. To install all Async Professional files (in either OOP or non-OOP mode) requires about 8MB of free disk space.
5. At least one standard com port.

Optional:

6. Borland's Turbo Assembler (TASM) 1.0 or higher, or Microsoft's Macro Assembler (MASM) 5.0 or higher. An assembler is needed only if you wish to modify any of the assembly language routines.
7. TurboPower Software's Turbo Professional 5.0 or later (5.20 or later if using protected mode) Object Professional 1.0 or later (1.20 or later if using protected mode). Portions of Async Professional are designed to take advantage of facilities found in these two products if they are available. In addition, the TERMWIN unit requires Object Professional, and the streams support in Async Professional is based on the streams system in Object Professional.
8. A Microsoft-compatible mouse.

There are also certain assumptions this documentation makes of the programmer. It provides no tutorial background on Pascal programming; the user is expected to have an understanding of Turbo Pascal syntax, unit structure, and use of the compiler itself.

Although Async Professional handles the details of asynchronous communications, the documentation assumes a basic understanding of the issues of serial communications, modem operation, terminal emulation and file transfer protocols. Chapter 2 provides a brief explanation of serial communications in general; the other topics receive similar introductions in their respective chapters. A glossary in Appendix A defines various terms used throughout the manual.

Despite these brief explanations, this documentation cannot serve as a replacement for your modem manual, for the National Semiconductor Data Sheets on various UARTs, or for the various protocol specifications. Generally, the information provided in our documentation is sufficient to use the Async Professional routines. To go beyond these routines and capabilities, you will probably need to refer to some of the documentation mentioned above.

B. Installation

Async Professional is distributed on dual media: 5.25" 1.2MB floppy diskettes and 3.5" 720KB diskettes. The number of diskettes of each media type is subject to change. The first 5.25" diskette is labeled DISK5-1; subsequent diskettes are labeled DISK5-2, and so on. The first 3.5" diskette is labeled DISK3-1; subsequent diskettes are labeled DISK3-2, and so on. Async Professional can be installed directly from either set of diskettes or you can copy the complete set of files to your hard disk and run the installation from there.

We'd like to draw your attention to several files in particular, all on the first disk:

READ.ME

Read this file before installing. It contains last minute information that may affect how you install the product.

READ.1ST

This file is in the archive FILES1.LZH, which the installation program dearchives for you. Print and read this file before using the product. It contains the latest updates to this documentation and may contain new features, clarifications, or corrections.

FASTUPD.xxx

Describes changes in this version of Async Professional compared to the most recent previous one. This file is of particular use to Fast Update Plan subscribers. The characters "xxx" specify the current version number, for example, 201 means version 2.01.

LHA.EXE

A freeware compression/decompression utility used to compress and decompress files with extension LZH. Type LHA to get a summary of its options. LHA.HLP contains complete documentation of LHA.

PACKING.LST

A complete list of all the files on all the diskettes, including the contents of the archives.

PRINTDOC.EXE

This file is in the archive FILES1.LZH, which the installation program dearchives for you. PRINTDOC.EXE prints text files with pagination, margins, and headers. This small utility is useful for printing files like READ.ME. Type PRINTDOC to get a list of options.

The Installation Process

INSTALL is an interactive tool that makes it very easy for you to dearchive the contents of the archives on the Async Professional disks. With it, you can select only as many Async Professional files as you actually need.

If you're installing directly from diskettes, insert the first disk in drive A: and enter

A:INSTALL

The main window is displayed:

Async Professional Installation Program

Install Options

Source	[A:\	]
Destination	[C:\APRO\	]
Calling Convention	[Object-Oriented Interface	]
File Groups	[Base Communications, Protocols, Fax, Compression,]	]

<Ctrl><Enter> to begin install, or Enter path for source

'Source' refers to the disk you will install from (where the compressed LZH files are located). This defaults to the drive where INSTALL.EXE was located.

'Destination' specifies the directory where Async Professional will be installed. This is where the decompressed files will be placed. The default is C:\APRO if you are installing from diskettes, or X:\APRO if you are installing from hard drive X. If the directory does not exist, INSTALL creates it for you.

If you want to change the default directories, use the cursor keys to position the highlight bar over the directory, and make the necessary changes.

The 'Calling Convention' option allows you to choose whether to install the modules required for the object-oriented interface, or the procedural interface, or both. Move to the 'Calling Convention' field and press <Enter>. The following popup pick list is displayed.

Interface

- ▶ Object-Oriented Interface
- Procedural Interface
- Both

The object-oriented interface is selected by default. You can cursor up and down and toggle the selection of the interface by pressing <Space>.

Now move to the 'File Groups' field and press <Enter>. A popup pick list is displayed and you can choose which file groups to install:

Groups

- ▶ All
- ▶ Base Communications
- ▶ Protocols
- ▶ Fax
- ▶ Compression
- ▶ OPRO Demos (source)
- ▶ Help
- ▶ Bonus

Selected file groups are marked with '▶'. If you want to install all of Async Professional, simply press <Enter>. Otherwise, you can cursor up and down and toggle the selection of file groups by pressing <Space>. As you select and deselect file groups, any prerequisite file groups are automatically selected or deselected. Select All to select all file groups.

Base Communications

These basic communication functions are required to use the protocols, fax capabilities, or Async Professional demonstration programs.

Protocols

This installs the modules that implement Xmodem, Ymodem, Zmodem, Kermit, B+, and ASCII protocols.

Fax

This installs the modules that control a faxmodem in sending and receiving faxes.

Compression

This installs the modules that provide compression and decompression of files in both LZH and ZIP formats.

OPRO Demos (source)

These demonstration programs require Object Professional. Do not choose this file group unless you also have Object Professional.

Help

'Help' installs POPHELP.EXE, MAKEHELP.EXE, and all the text files used in the on-line help system. The Borland Pascal 7.0 help file (APRO.TPH) is also installed. See §1.G for more information on installing and using either POPHELP or the Borland Pascal 7.0 help file.

Bonus

This file group contains modules that are provided to you on an as-is basis. These modules are not documented in this manual, but are referred to when appropriate. The file named READ.ME! gives a brief description of each bonus file.

When you are finished specifying install options, press <Ctrl><Enter> to start the installation. INSTALL displays the install progress in a window.

If all of Async Professional is installed, your directory tree will look something like the following:

```
└─APRO
   ├──HELP      POPHELP, MAKEHELP, APRO.HLP, APRO.TPH, help text files
   ├──BONUS     Bonus files subdirectory
   └──EXAMPLES  Examples subdirectory
```

The base directory, \APRO by default, contains most of the Async Professional modules, the READ.ME file, and the FASTUPD.xxx file (where xxx is the version number).

If you'll be working from another directory while using the Async Professional units, be sure to add your \APRO subdirectory to the IDE's search paths for Include Directories, Unit Directories, and Object Directories. If you use the command line compiler, add the following lines to your configuration file:

```
-Ud:\apro
-Od:\apro
-Id:\apro
```

where d: specifies the drive where your \APRO directory is located.

Note that precompiled units are not provided, since they are dependent on which version of Turbo Pascal you're using and how you configure Async Professional (see the following section for configuration information). In any case, simply ask the compiler to Make your program and it creates the required Async Professional TPU files automatically.

Two make files (APRO.MAK and APROO.MAK) are provided for use with Borland's standalone MAKE utility. There's no requirement for you to use these files; they're provided primarily as a complete specification of the interdependencies of the files, and also as a convenient way to build all the units and object files at once (APRO.MAK has the specifications for the procedural units and programs, APROO.MAK has the OOP specifications). You will probably need to edit the appropriate make file to specify the location of your compiler and assembler. The make file contains instructions on how to perform any needed modifications. To run APRO.MAK, type the following DOS command line (note that the -f must be lowercase):

MAKE -fAPRO.MAK

Modules Provided

Here's a brief overview of all the units provided by Async Professional:

APTIMER

A unit containing BIOS timing functions (with a one clock tick resolution). This is used by many other Async Professional units, but is provided as a separate unit so that your program can use its routines as well.

APPORT

Required by all communications programs. This is an abstract unit that provides all the data and procedure type definitions required by other parts of Async Professional.

APMISC

Also required by all communications (and dearchiving) programs. This unit contains a variety of general purpose items (such as the error constants, date routines, CRC and checksum routines).

APUART

Contains the interrupt-driven serial I/O routines that most communications programs will use.

APFOSSIL

Contains routines using a FOSSIL communications driver.

APDIGI14

Contains routines using a DigiBoard communications adapter driver.

APINT14

Contains routines using the BIOS Int \$14 serial communications standard.

APCOM

The procedural application programming interface.

APANSI

The procedural routines for implementing ANSI terminal emulation.

APMODEM

The procedural routines for working with Hayes-compatible modems.

APCOUR

The procedural routines for working with U.S. Robotics 9600 BPS modems.

APMCOM

The procedural routines for working with MicroCom 9600 BPS modems.

APABSPCL

An "abstract" unit that contains the base procedural routines for working with protocols.

APXMODEM, APYMODEM, APZMODEM, APKERMIT, APBPLUS, APASCII

The procedural routines for implementing the Xmodem, Ymodem, Zmodem, Kermit, B+, and ASCII protocols.

APFAXCVT, APABSFAX, APFAX12, APFAXCAS

The procedural routines for converting, sending, and receiving faxes.

APARCHIV, APLZH, APZIP

Procedural facilities for working with archives in both LZH and ZIP formats.

OOCOM

The objects for the basic application programming interface.

OOMODEM

The objects for working with Hayes-compatible, U.S. Robotics, and MicroCom modems.

OOEMU

The basic terminal emulation object and an ANSI emulator object.

OOABSPCL

The AbstractProtocol from which all other protocol objects are derived.

OOXMODEM, OOYMODEM, OOZMODEM, OOKERMIT, OOBPLUS, OOASCII

The Xmodem, Ymodem, Zmodem, Kermit, B+, and ASCII protocol objects.

OOFAXCVT, OOABSFAX, OOFAX12, OOFAXCAS

The objects for converting, sending, and receiving faxes.

OOARCHIV, OOLZH, OOZIP

The objects for working with archives in both LZH and ZIP formats.

TERMWIN

A window object that can display serial input/output within a window. This is derived from an Object Professional class.

Demonstration Programs Provided

Async Professional provides a variety of demonstration programs. Of these, only four are provided in executable form: COMTEST (to ensure that it's compiled with all options enabled), OOPCOM, RFAXO, and TPFAX (since they require Object Professional). The rest of the demonstration programs can easily be compiled from the supplied source files.

UARTID

Identifies all UARTs (serial ports) that physically exist within your machine. Can also be used to enable or disable the UART's FIFO buffers. See §1.E for a description of UARTID.

COMTEST/COMTESTO

A general purpose diagnostic program. It provides simple input and output functions as well as the status of various UART bit values, hardware and software flow control and input/output buffers. COMTEST uses procedures; COMTESTO uses objects. See §1.E for a description of COMTEST/COMTESTO.

SIMPCOM

A simple general-purpose communication program written using only the facilities of Async Professional. Written using the OOP units; includes protocols and terminal emulation with simple modem support. See §1.F for a description of SIMPCOM.

OOPCOM

A full-featured communication program written with Async Professional and Object Professional. Demonstrates how to integrate TerminalWindows into an Object Professional program. Also shows how to create a Dialer object and a library of modem objects. OOPINST.EXE is the installation program for OOPCOM. See §1.F for a description of OOPCOM.

FX/FXO

A general purpose file transfer program. Shows examples of nearly every Async Professional protocol feature: all protocols, status routines, logging, file management, and transmit file lists. FX uses procedures; FXO uses objects. See §7.J for a description of FX/FXO.

BACK/BACKO

TSR programs to do background file transfer. Conditional defines allow you to select the protocol to be used. See §7.J for a description of BACK/BACKO and TTERM/TTERMO.

TERMTEST

A simple demonstration of the TerminalWindow object. See §9.F for a description of TERMTEST.

SIMPSND/SIMPSNDO

Procedural and OOP programs for sending a fax. See §10.G for a description of SIMPSND/SIMPSNDO.

SIMPRCV/SIMPRCVO

Procedural and OOP programs for receiving a fax. See §10.G for a description of SIMPRCV/SIMPRCVO.

CVT2FAX/CVT2FAXO

Procedural and OOP programs for converting files to fax format. See §10.G for a description of CVT2FAX/CVT2FAXO.

SHOWFAX/SHOWFAXO

Procedural and OOP programs for displaying fax files. See §10.G for a description of SHOWFAX/SHOWFAXO.

PRNFAX/PRNFAXO

Procedural and OOP programs for printing fax files to PCL5-compatible or Epson printers. See §10.G for a description of PRNFAX/PRNFAXO.

FAX2PCX/FAX2PCXO

Procedural and OOP programs for converting Async Professional fax files to PCX format. See §10.G for a description of FAX2PCX/FAX2PCXO.

RFAX/RFAXO

Procedural and OOP programs for a background TSR fax receiver. See §10.G for a description of RFAX/RFAXO.

FAXSRVR/FAXSRVRO

Procedural and OOP programs that implement a dedicated fax receiver and printer. See §10.G for a description of FAXSRVR/FAXSRVRO.

TPFAX

An OOP program that implements full-featured fax management. See §1.F for a description of TPFAX.

LZH/LZHO

Procedural and OOP programs for compressing files into an LZH archive. See §11.E for a description of LZH/LZHO.

LZHV/LZHVO

Procedural and OOP programs for viewing the contents of LZH archives. See §11.E for a description of LZHV/LZHVO.

LZHX/LZHXO

Procedural and OOP programs for extracting files from LZH archives. See §11.E for a description of LZHX/LZHXO.

ZIP/ZIPO

Procedural and OOP programs for compressing files into a ZIP archive. See §11.E for a description of ZIP/ZIPO.

ZIPV/ZIPVO

Procedural and OOP programs for viewing the contents of ZIP archives. See §11.E for a description of ZIPV/ZIPVO.

ZIPX/ZIPXO

Procedural and OOP programs for extracting files from ZIP archives. See §11.E for a description of ZIPX/ZIPXO.

C. Configuration

Before using Async Professional units, you need to understand and perhaps modify various conditional defines in the file APDEFINE.INC. This file is a Turbo Pascal include file that is included into all Async Professional units. To modify it, load it into a text editor and find the appropriate conditional symbol. We use the convention of inserting a period between the "{" and the "\$" of a conditional define in order to deactivate that define. For example, the following define is active:

```
{ $DEFINE UseOPro }
```

The deactivated form of the same define is

```
{ . $DEFINE UseOPro }
```

With the period in place, the compiler sees the line as a plain comment, which has no effect on compilation.

After modifying APDEFINE.INC, save it to disk and use the compiler to Make your applications for the changes to take effect.

Device Layer Selection

Async Professional comes with four ready-to-use device layers: APUART, which provides interrupt-driven serial I/O up to 115K baud; APFOSSIL, which provides device layer support for standard FOSSIL device drivers; APDIGI14, which supports DigiBoard intelligent multiport boards; and APINT14, which provides the simple serial services available through BIOS Int \$14. More details about what a device layer is and why you might want to use a different one are provided later. For now, you'll probably want to stick with the default device layer of APUART. If, however, you do want to use a different device layer, change one or more of the following lines in APDEFINE.INC:

```
(Enable one or more of the following defines. These specify which "device layer" will be included in your Async Professional programs.)
```

```
{ $DEFINE UseUart }  
{ . $DEFINE UseInt14 }  
{ . $DEFINE UseFossil }  
{ . $DEFINE UseDigi14 }
```

These defaults activate the device layer APUART and omit all other device layers. This is the desired configuration for most programs. When you need to use a different device layer, uncomment its \$DEFINE and comment out UseUart.

It is possible to have more than one device layer active at one time (which would allow your programs to choose the proper device layer at runtime). See the demonstration programs COMTEST and COMTESTO for some examples of this.

APUART includes an assembly language interrupt service routine (ISR) to service serial port interrupts. The ISR supports the following Async Professional options: hardware flow control, software flow control, status buffering, and event logging (these features are described in detail in later chapters). The ISR is assembled in seven different configurations (files APUART1.OBJ through APUART7.OBJ). You'll need to pick the configuration that best matches your needs. If you're not sure what options you need, just stick with Standard; you can always change it later if you find you need a different configuration. The following extract from APDEFINE.INC shows the various configuration choices:

{ \$DEFINE Standard }

{ Valid Alternatives:

- Standard - Automatic software and hardware flow control
- StandardLogging - Standard plus EventLogging
- Status - Standard plus StatusBuffering
- StatusLogging - Status plus EventLogging
- HWOnly - Automatic hardware flow control only
- SWOnly - Automatic software flow control only
- Basic - No options at all (no flow control, no status, no logging)
- UserDefined - User-customized options

The following provides more detail on each of the supported configurations.

Standard

The standard configuration of ISR options includes both hardware and software flow control, but no status buffering or debugging support.

StandardLogging

This configuration includes both hardware and software flow control plus the debugging facility known as EventLogging. This slows down the ISR considerably, so use this option only while debugging your program.

Status

The Status configuration includes both hardware and software flow control plus the feature known as Status Buffering. When Status Buffering is enabled, each received character has an associated status byte with the exact status of that particular byte (OK, overrun error, parity error, or framing error).

StatusLogging

This configuration includes hardware and software flow control, the Status Buffering feature and the EventLogging feature. Use this configuration only when you need the Status Buffering feature and you are also doing some debugging that requires EventLogging. This is the largest and slowest of all ISR configurations.

HWOnly

This is a small and fast configuration. The only feature it provides is automatic hardware flow control. Use this configuration when you don't need software flow control and you're not doing any debugging.

SWOnly

This is another small and fast configuration. The only feature it provides is automatic software flow control. Use this configuration when you don't need hardware flow control and you're not doing any debugging.

Basic

This is the smallest and fastest configuration. It's the basic ISR without any flow control, status buffering, or debugging features.

UserDefined

If none of the above configurations fit your exact needs, you can customize APUART directly (by enabling and disabling various conditional assembly symbols) and assemble it to APUART9.OBJ. This OBJ file is linked in whenever UserDefined is specified. Further information on this subject is not provided, since anyone interested in such customizations probably won't have any trouble figuring out how to do so (see APUART.ASM to start).

APDEFINE.INC provides a few other conditional defines that you might want to customize. Here's a brief description of those symbols:

UseOOP

UseOOP is not defined by default. When defined, it removes the global procedure pointers required by the non-object-oriented parts of the base library. Leaving this undefined won't bother the OOP routines since they don't use the procedure pointers. However, if you are using the OOP format of Async Professional, you can activate this define to save a few hundred bytes of code.

Tracing

Tracing is a debugging tool built into Async Professional. It is not defined by default. Activate this define and recompile Async Professional whenever you need to debug your programs with Tracing. A reasonable approach might be to turn it on during the early stages of development (so it's always handy) and turn it off as your program nears completion.

UseOPro

UseOPro, which is not defined by default, assumes that you have the units of TurboPower Software's Object Professional available when compiling Async Professional. When defined, UseOPro causes Async Professional to omit some of its general-purpose code (several thousand bytes worth) and use routines from the following Object Professional units: OPCODE, OPROOT, OPSTRING, OPINLINE, and OPDATE.

UseTPro

UseTPro, which is not defined by default, assumes that you have the units of TurboPower Software's Turbo Professional available when compiling Async Professional. When defined, UseTPro causes Async Professional to omit some of its general purpose code (about 3000 bytes worth) and use routines from the following Turbo Professional units: TPMEMCHK, TPSTRING, TPINLINE, and TPDATE.

AutoDeviceInit

AutoDeviceInit, which is defined by default, specifies that Async Professional should automatically activate a device layer. Specifically, it means that the initialization block of a device layer should call its own ActivateXxx procedure (e.g., APUART's initialization block would automatically call ActivateApUart). Under rare circumstances, very simple communications applications may get smaller (by 1 or 2KB) if you don't define AutoDeviceInit. Note that this means you must manually initialize the global procedure pointers that you will be using. AutoDeviceInit is forced off when UseOOP is defined (since the OOP interface layer doesn't use the global procedure pointers).

LargeComNameSet

LargeComNameSet is not defined by default. Activate this define only if you use a device layer, such as Digi14Port, that supports more than the standard 8 com ports. Defining LargeComNameSet increases the size of ComNameType (to Com1..Com36) and expands all arrays of ComNameType accordingly.

UsePmodeDLL

This controls whether the procedural portion of Async Professional is built as TPU/TPP files or as two protected mode DLLs (APCOMD for the device and interface layers, and APPROTD for all protocols).

BindFaxFont

This controls whether APFAX.FNT, the font file used to convert ASCII text files to fax binary files, is bound to the APFAXCVT/OOFAXCVT units or not. Binding the font to the TPU results in fast loads from LoadFont, but adds about 16KB of code.

Async Professional also provides some symbols that are automatically defined whenever the selected ISR configuration includes the associated feature. You can use these defined symbols in your program to conditionally compile portions of your source code. For example, you might want to surround your calls to `InitEventLogging` and `DumpEvents` with the `{ $IFDEF EventLogging }` directive. This deactivates your debugging code unless you are using an ISR configuration that includes event logging.

EventLogging

`EventLogging` is defined when `StandardLogging` or `StatusLogging` is defined.

StatusBuffering

`StatusBuffering` is defined when `Status` or `StatusLogging` is defined.

UseHWFlow

`UseHWFlow` is defined when `Standard`, `StandardLogging`, `Status`, `StatusLogging`, or `HWOnly` is defined.

UseSWFlow

`UseSWFlow` is defined when `Standard`, `StandardLogging`, `Status`, `StatusLogging`, or `SWOnly` is defined.

Compiler Options

Each unit provided with Async Professional begins with source code similar to the following:

```
{ $S-,R-,V-,I-,B-,F+,O+,A- }
```

```
{ Conditional defines that may affect this unit }
```

```
{ $I APDEFINE.INC }
```

The compiler directives specify the default settings for which the unit was written. `APDEFINE.INC` is included to allow specification of user-defined options. Note that you may modify `APDEFINE.INC` to specify compiler options and conditional symbols.

All Async Professional units can be overlaid except `APUART`, `APFAXCVT/OOFAXCVT` (if `BindFaxFont` is defined in `APDEFINE.INC`), `APARCHIV/OOARCHIV`, `APZIP/OOZIP`, and `APLZH/OOLZH`. Please keep performance considerations in mind when choosing which units your application will overlay.

The following units have initialization blocks. Overlaying such a unit requires that Turbo Pascal's overlay manager be initialized before the initialization block is executed. See the Turbo Pascal manual for more information.

<code>APANSI</code>	<code>APCOM</code>	<code>APDIGI14</code>
<code>APFAXCVT</code>	<code>APFOSSIL</code>	<code>APINT14</code>
<code>APMISC</code>	<code>APPORT</code>	<code>OOCOM</code>
<code>OOFAXCVT</code>	<code>TERMIN</code>	

When using library packages such as Async Professional, remember that the Turbo Pascal compiler has a finite capacity. When you write a short program that uses units provided by others, you may forget that the compiler is actually processing lots of code provided by the units listed in the `uses` statement. To maximize the compiler's capacity, you should apply the following advice. (Because Borland Pascal has a much higher capacity, the following does not apply and you can usually turn debug information on everywhere.)

1. In the `TURBO.EXE` integrated environment, apply the following settings:

Compile Destination
Debug Information

Disk
Off

Local Symbols	Off
Link Buffer	Disk
Integrated Debugger debug info	Off
Standalone Debugger debug info	Off

Although these settings may lead you to believe that you can't use a debugger, that's not true. The important goal is to add debug information only to those units where you need to trace, not in every unit.

2. If you outgrow the capacity of the integrated environment, use the command line compiler TPC.EXE instead. When using TPC, make the following options part of your TPC.CFG file (see the Turbo Pascal manual) or part of your command line:

```
/SD- /$L- /L /M
```

These options turn off debug information, force link buffering to disk, and perform a make by default. Again, the key to compiling with debug information is to add such information to only those units where you need it and to strip the information after you're finished with it. To compile a single unit with debug information, use the following command line:

```
TPC /SD+ /$L+ /L UnitName
```

To compile a program with debug information for Turbo Debugger, use the following command line:

```
TPC /SD+ /$L+ /L /V ProgName
```

Do not use the /V option regularly, since it causes the compiler to append debug information to your program's EXE file, which will bloat it significantly.

3. If you outgrow even the capacity of TPC.EXE, consider using TPCX.EXE protected mode compiler of Turbo Pascal 6.0 or BPC.EXE of Borland Pascal 7.0. These compilers use up to 15 megabytes of extended memory to provide almost unlimited capacity.

There are various other tricks for improving compile capacity, although in most cases they offer second-order improvements compared to the items already listed. Keep the following in mind:

- Unload as many TSRs and device drivers as possible.
- Turn off range and stack checking {\$R-, S-}
- Extract all units from TURBO.TPL by using TPUMOVER. In this case, the compiler will load other units such as SYSTEM.TPU and DOS.TPU only when the source code calls for them.
- Keep the main program as small as possible and instead put all code into units. After following this advice, your main program will use a single main unit and contain a single call to the one "Main" routine interfaced by that unit.
- Minimize the number of identifiers interfaced by every unit.
- Carefully set the defines in files such as APDEFINE.INC to activate only the library features that you need.

D. Organization of this Manual

This manual for Async Professional is divided into twelve chapters. Chapter 1 covers introductory topics: what Async Professional does, how to install it, how the diagnostic and primary demonstration programs work, how to use the online help facility, and what to check before calling for technical support. The final section of the chapter is the purchase agreement, which describes your rights and responsibilities as an owner of Async Professional.

Chapter 2, also introductory, discusses the basic issues involved in writing programs that perform asynchronous communications, as well as the architecture of the Async Professional library. Even if you are experienced in writing communications programs, you should read this chapter before attempting to use any of the units in the library.

Chapters 3-11 provide detailed discussions of the units that comprise the library. These chapters begin with a broad overview of all the units covered in the chapter, followed by a series of sections devoted to the individual units, each of which ends with a reference section documenting all of the public identifiers interfaced by the unit. In several cases, a chapter ends with a discussion of relevant example programs.

Chapter 12 contains four appendices, which cover important topics that don't fit neatly into any of the preceding chapters. Appendix A provides a glossary of terms that every Async Professional user should be familiar with. Appendix B is a list of all the error codes that can be generated by the various routines in the library. Appendix C provides tips on debugging programs that perform asynchronous communications. And Appendix D shows serial port diagrams for 9-pin and 25-pin connectors.

Two indexes are provided: a conventional subject index and a separate index of identifiers (procedures, functions, methods, types, constants, and variables).

The reference sections of the manual explain the contents of each unit in detail. Routines (procedures, functions, methods) are documented one or two to a page wherever possible. Due to space and cost considerations, however, you may find more than two functions per page where they will fit neatly. In any case, the names of all routines on the page are always printed at the top of the page in bold type. Since the routines are listed in alphabetical order, you can flip through the manual to find a specific routine fairly quickly. The unit name is shown on the last line of each page so that you can flip to find those as well.

Note that in several cases a reference section is used to document two units that perform largely identical functions, one using objects and one not. For example, the reference section in §6.C covers both APMODEM and OOMODEM. This reference section begins with the documentation for all constants, types, and variables interfaced by the two units. Those that are identical in both units are listed first, under "Shared Declarations." Those that are unique to APMODEM are listed next, under "APMODEM Declarations," and those unique to OOMODEM are listed last, under "OOMODEM Declarations." Immediately following the declarations section are the entries for all interfaced routines/methods, which appear in alphabetical order. In cases where both units have a routine/method with the same name and purpose, there is a single entry, which points out any noteworthy differences between the two.

Each reference section entry for a procedure, function, or method uses the following basic format:

Syntax

The Pascal declaration for the routine, showing a complete parameter list. If the routine exists in both OOP and non-OOP versions, then both declarations are given.

Purpose

A one-sentence description of the routine's purpose.

Description

Details about the parameters to the routine, its operation, possible side-effects, and the error conditions it reports.

Examples

Code fragments or minimal programs showing how to use the routine.

See Also

Related routines/methods. If the routine is in the same object or unit, it is listed just by name. If it is a routine in another unit, it is shown as RoutineName (UNITNAME). If it is a method of another object, it is shown as ObjectName.MethodName.

Keep in mind that there are two supplements to the documentation in this manual: the example programs and the READ.1ST file. We've put a lot of effort into preparing complete example programs, with commented source code, because many people learn best by example. The introductory example programs, those listed in the text of the manual, are also provided in source form on disk so that you don't have to type them in. The name of the corresponding source file is in a comment on the right side of the first line of the program.

The READ.1ST file answers common questions, describes new features of the software, corrects errors in the manual—all very important stuff to know. Please be sure to read this file. It's a good idea to print it out, so that a few months from now you'll find it sticking out from the edge of your manual and remember to look at it again.

E. Diagnostic Programs

Async Professional includes a couple of simple diagnostic programs that you can use to verify that your hardware (including the UART, cabling, and the remote device itself) is functioning properly. The first of these programs is UARTID. Its job is simply to verify the existence of a UART within your machine. The second is COMTEST, which provides simple serial I/O capabilities and some UART status information.

1. UART Identification Program: UARTID

This program checks your hardware and reports how many UARTs (serial ports) it found. Here's a sample report:

```
COM1 is an 8250A or 16450 [known to BIOS]
  Line parameters: 2400,None,8,1
COM2 is a 16550A [known to BIOS] [FIFO is On]
  Line parameters: 19200,None,8,1
COM3 is an 8250A or 16450 [known to BIOS]
  Line parameters: 2400,None,8,1
COM4 is an 8250A or 16450 [known to BIOS]
  Line parameters: 2400,None,8,1
```

UARTID found four serial ports in this machine: COM1-COM4. In each case, UARTID reported the type of UART, whether or not that port was listed in the BIOS data area, and the current line parameters for that port.

The type of UART is generally not of interest, except that only 16550 UARTs provide onboard FIFO buffers. Note that for COM2, UARTID also listed the state of the FIFO buffer (it does this only for 16550's, since only they have FIFO buffers).

If the UART address is listed in the BIOS data area, then UARTID report includes the phrase "[known to BIOS]." The BIOS data area is filled in during the bootstrap process. Some BIOS's check only for COM1 and COM2 and ignore all other serial ports. This isn't a problem unless you are running a communications program that plays things very safe and won't open a serial port unless it's listed in the BIOS data area.

UARTID expects a command line of the following form:

```
UARTID [Options]
```

where Options are one or more of the following:

1-8	Portnumber for toggling FIFO
On/Off	New FIFO status
Ln	Trigger level for FIFO use (n=1,4,8,14)
S	Safe mode
?	Display help screen

The first three options can be used together to enable or disable the FIFO buffer on a 16550 UART. To do so, you would supply the port number, the new status, and a trigger level. For example, to turn on COM2's FIFO buffers with a trigger level of 8, specify:

```
UARTID 2 on l8
```

Turn it off with:

```
UARTID 2 off
```


While running UARTID, you might notice that the trigger level is displayed when you first turn the FIFO on but never again. That's because it's not possible to determine an existing trigger level.

The S option specifies "safe mode." In this mode, UARTID will report only on UARTs that already exist in the BIOS data area. This defeats, to some extent, the original purpose of UARTID, in that it is supposed to ignore the BIOS data area and do its own search for UARTs. The main reason for the safe mode is to overcome a problem with this searching technique and ARCNET network boards. The problem is that the default address for an ARCNET network board overlaps the standard address of COM4, and reading from any of the default COM4 I/O addresses will disrupt the ARCNET board and probably crash the network shell.

Such problems can occur with *any* hardware in your machine that conflicts with the address of a com port. The ARCNET case is known and is likely to exist in some of your customer's machines as well, so you may also want to consider a "safe" option in your program.

2. Communications Test: COMTEST/COMTESTO

Communications programming brings along some occasionally frustrating hardware problems. Is the cable wired correctly? Is the cable broken? (or even plugged in?) Is the device attached to the serial port sending anything? receiving anything?

Often, in the early stages of setting up a new piece of hardware (instrument, modem, printer, or whatever) it's nice to be able to rely on tried-and-true software to test out the hardware. That's where COMTEST comes in. It provides simple transmit and receive functions (in separate windows) and status information about the UART, flow control, and I/O buffers.

COMTEST also implements the standard debugging tools of Async Professional (Tracing and EventLogging) and adds data capturing and hex mode displays. In short, COMTEST should be the first tool you turn to when your main application "isn't getting anything" or when you need some detailed information (a Trace, EventLog, or capture buffer) showing what your remote device is doing.

Note that everything said here about COMTEST applies equally well to COMTESTO, the OOP version. There's really no functional difference between the two. However, since these test programs are also examples of using Async Professional routines, we wanted to make sure that both procedural and OOP techniques were illustrated.

The COMTEST screen looks something like this:

Line Ctrl/Status		Modem Ctrl/Status		Async Professional Port Setup		ComTest 2.00 Buffer Status		Misc.	
WSLO	DR	DTR	DCTS	ComPort	COM1	MaxSize	5000	Flow	Off
WSL1	OE	RTS	DDSR	Baud	1200	InFree	5000	Hdw	Off
STB	PE	OW1	DRI	Parity	None	InUsed	0	Xsent	
PEN	FE	OW2	DDCD	Data	8	OutFree	5000	XRcvd	
EPS	BI	LOOP	CTS	Stop	1	OutUsed	0	Overrun	
STK	THRE		DSR					Parity	
BRK	TEMT		RI					Frame	
DLAB	FERR		DCD					Break	
Receive Window									
Transmit/Message Window									
F1-Help F2-Xmit Test F3-DTR F4-RTS F5-Clear F6-Hex Mode									

The top window shows all the status information. The columns labeled "Ctrl/Status" show the current state of the status and control bits of the line and modem registers. (See Chapter 2 for more information on the meanings of these bits. Note that COMTEST lists the bit names given on the UART data sheets rather than the more meaningful names used in Chapter 2.) If a particular bit is currently set, it is displayed in intense video; otherwise it is dim.

The next column, "Port Setup," shows the current port and line parameters. These should be fairly self-explanatory.

Next is the "Buffer Status" column. Again, these values should be self-explanatory. You'll probably care about these values only if you are testing the flow control mechanism of an attached device.

The last column, "Misc.," shows the current status of hardware and software flow control. If COMTEST has sent an Xoff, it displays "XSent." It removes that flag when it sends an Xon. If COMTEST receives an Xoff, it displays "XRcvd." It removes that flag when it receives an Xon. (COMTEST also displays the current status of hardware handshaking in the Modem Control and Status bits in columns three and four.) The "Misc." column is also used to display the various line error conditions (Overrun, Frame, Parity, and Break). These flags remain set until you clear them by pressing <F5>.

The middle window is the Receive Window. All characters received are displayed in this window. It wraps to the next line whenever it receives a CR/LF or the current line reaches the end of the display. You can toggle the display of new characters between ASCII and hex by pressing <F6>.

The bottom window is the Transmit/Message Window. This is kept separate from the Receive Window so it is clear which data was typed in from the keyboard and which was received from the remote. This window also doubles as the message window in those few cases where COMTEST needs to display a message.

The last line displays most of the commands available in COMTEST:

<F1>

Pops up a small help screen with one-line descriptions of the various COMTEST commands.

<F2>

Performs a transmit test. The default transmit test is simply to send the string '1234567890' ten times (for a total of 100 characters).

<F3>, <F4>

Toggle the DTR and RTS lines, respectively. Use the Modem Ctrl/Status column to note the current state of these lines.

<F5>

Clears any pending line error flags (such as a parity error).

<F6>

Toggles the received data display between ASCII and hex mode.

<F9>

Invokes the "current test." COMTEST doesn't supply such a test. However, a safe modification of the COMTEST source code would be for you to add some test specific to your needs in the procedure called CurrentTest.

COMTEST also supports several <Alt> commands for controlling various options:

<AltC>

Next com port (cycles through all known ports).

<AltB>

Next baud rate (cycles from 1200 to 115K).

<AltD>

Next data bit setting (cycles from 5 to 8).

<AltS>

Next stop bit setting (1 or 2).

<AltP>

Next parity setting (None, Even, Odd, Mark, Space).

<AltF>

Toggles automatic Xon/Xoff flow control.

<AltH>

Toggles automatic hardware handshaking.

<AltA>

Toggles capturing of received data. Captured data is stored in COMTEST.CAP.

<AltT>

Toggles Tracing of PutChar/GetChar calls. Trace information is stored in COMTEST.TRC.

<AltE>

Toggles EventLogging of low-level interrupts. Logged data is stored in COMTEST.LOG.

<AltX>, <Esc>

Exit from COMTEST.

Turning off the Tracing, EventLogging, or Capture option causes the appropriate file to be written and closed. If you exit the program with any of these options on, the appropriate file is written and closed. Turning on one of these options causes the associated file to be overwritten without warning.

COMTEST expects a command line of the following form:

COMTEST [Options]

where options are one or more of the following:

/B Baudrate	1200-115200 [Default=1200]
/C Comport	1-36 [Default=1]
/M	Force monochrome mode
/I	Use an Int14 device
/F	Use a FOSSIL device
/D	Use a Digi14 device
/X	Skip BIOS check when opening ports
/?	Display list of options

The /B and /C options are fairly self-explanatory. Note that the option letter and the option value must be separated by one space (e.g., /C 2).

Use /M to force COMTEST to use monochrome attributes on a color display. You'll usually need this on laptops that report that they are color monitors but do a poor job of mapping colors to grey scales.

The /I option lets you try COMTEST using the Int 14 interface (not very exciting but shows that it can be done).

The /F option lets you try COMTEST using a FOSSIL device.

The /D option lets you try COMTEST using a Digi14 device.

The /X option tells COMTEST not to verify that a com port is in the BIOS data area before opening it. Note that this is just the opposite of UARTID's "safe mode" option. By default, COMTEST will not attempt to open a port unless it finds that port address in the BIOS data area. When you specify /X, COMTEST skips this check and attempts to open every port between 1 and 4 (or 1 and 8 on a PS/2). If you have other hardware using the default addresses of those ports, the results are unpredictable.

F. Demonstration Programs

Async Professional comes with two examples of general purpose communications programs written using the library. The first and less sophisticated of the two, SIMPCOM, uses only routines found in Async Professional and the Turbo Pascal runtime library. The second, OOPCOM, takes advantage of the advanced user interface design tools found in Object Professional. You do not need a copy of Object Professional to run OOPCOM, since a compiled copy of it is provided, but you do need one if you want to modify or recompile OOPCOM.

1. A Simple Communications Program: SIMPCOM

As its name implies, SIMPCOM is a fairly simple communications program written with the object-oriented units in Async Professional (OOCOM, OOMODEM, OOABSPCL, OOXMODEM, OOYMODEM, OOZMODEM, OOKERMIT, OOASCII, OOBPLUS) and the APANSI unit. Although its user interface is not nearly as slick as OOPCOM, its looks are somewhat deceiving: this program has everything you need to log onto a BBS using a Hayes-compatible modem, capture the online session to a file, and upload or download files using any of the transfer protocols supported by Async Professional.

SIMPCOM passes all extended keystrokes (such as <RightArrow> or <Home>) through to the remote as a scancode and keycode. Treating extended characters like this is known as "doorway mode" in the BBS world. This permits SIMPCOM to use the full-screen editors provided by various bulletin board systems.

SIMPCOM expects a command line of the following form:

```
SIMPCOM [ConfigFileName]
```

The optional ConfigFileName parameter specifies a configuration file containing information like com port, baud rate, etc. (Further details on configuration files are given in the discussion of the <F2> command below.) If no configuration file is specified, SIMPCOM checks to see if the default configuration file, SIMPCOM.DAT, exists. If so, it is loaded automatically. If not, the default parameters (COM1, 2400 baud, no parity, 8 data bits, 1 stop bit) are used.

Once SIMPCOM is loaded, the following commands are available:

<F1>

Help. This command displays a popup window that shows the available commands and the keys they are assigned to.

<F2>

Set communications parameters. This command displays the following popup menu:

—Port Parameters—

Select the parameter to set:

B - to set baud	9600
P - to set parity	N
D - to set data bits	8
S - to set stop bits	1
C - COM port	COM2

Or press:

X - Save parameters
L - Load parameters

You can change the following parameters: baud rate (300-115200), parity (None, Odd, Even, Mark, Space), data bits (5-8), stop bits (1-2), and com port (Com1-Com4, Com1-Com8 on PS/2 machines). Selecting any of these options from the menu leads to a second menu that lets you change the individual setting. Press <Esc> to exit from any of these menus without making a selection.

The X option prompts you for the name of the file to store the parameters in. If you want the current settings to be loaded automatically when the program starts, use SIMPCOM.DAT as the name of your configuration file. To abort the operation, press <Esc>.

Once you've specified the name of your configuration file, SIMPCOM stores the following information in it: the com port in use; the settings for baud rate, parity, data bits, and stop bits; whether to use Tone or Pulse dialing, or the modem's default setting; and whether or not software and/or hardware flow control is enabled. (See the discussion of the <F5> command, below.)

When you select the L option, you are prompted for the name of a previously created configuration file to load. To abort the operation, press <Esc>. If the specified file exists, SIMPCOM opens the file and reconfigures itself based on the data stored in it.

<F3>

Dial modem. This command prompts you for a phone number to dial. To abort the operation, press <Esc>. If a phone number is specified and a connection is made, incoming text will appear in the window in the middle of the screen.

<F4>

Toggle file capture. When you first issue this command, you are asked for the name of the file to store captured data in. If you issued this command by mistake, press <Esc>. After you enter a name for the capture file, SIMPCOM checks to see if the file already exists. If it does, you are asked if you want to append the new data to it. If you answer No, you are asked if you want to overwrite the file. If you answer No to this question as well, the operation is aborted. Otherwise, SIMPCOM opens the capture file.

If capture is already on, you are asked to confirm that you want the capture file closed. If you answer No, the capture file remains open.

<F5>

Set options. This command displays the following popup menu:

```
Options
-----
Toggle assorted options:
A - ANSI mode           [On ]
O - Show Outgoing chars [Off]
C - Convert CR to CR/LF [Off]
I - Include directories [Off]
H - Hardware flow control [On ]
S - Software flow control [Off]
P - Default, Tone or Pulse [Def ]

T - Clear terminal window
```

The A option lets you toggle ANSI terminal emulation On or Off (it's On by default).

The O option lets you specify whether or not SIMPCOM should display Outgoing characters on the screen; this option is useful when communicating with a remote that doesn't echo received characters back to you (most BBS's do, so this option is Off by default).

The C option lets you specify that received carriage returns (CR's) should be translated to carriage return-line feed sequences (CRLF's); this option will generally need to be turned On whenever the display Outgoing characters option is needed.

The I option (Include directories) lets you specify whether or not the full pathname of a file should be sent to the remote when uploading files using a batch protocol. This option is Off by default.

The H option is used to toggle Hardware flow control On or Off; although it's On by default, keep in mind that it does nothing if you're using a modem that doesn't recognize the RTS signal (see the discussion of hardware flow control in §5.C).

The S option toggles Software flow control On or Off. Since software flow control is useful only if the remote recognizes Xon/Xoff signals, this option is Off by default.

The P option lets you specify whether to use Tone or Pulse dialing, or the modem's default setting.

<F6>

Hang up modem. You are asked to confirm that you want to hang up the modem. If you answer No, the online session continues without interruption.

<PgUp>

Upload a file. When you issue this command, SIMPCOM displays a popup menu asking you which transfer protocol to use: Xmodem, Xmodem 1K, XModem 1KG Ymodem, Ymodem G, Zmodem, Kermit, or ASCII. To abort the operation, press <Esc>. After making a selection, you are asked for the name of the file(s) to upload. If the protocol selected supports batch transfers, the filename can contain DOS wildcard characters: e.g., *.ZIP would transfer all ZIP files in the current directory. If a filename is entered, SIMPCOM begins transferring the file(s) to the remote. While the transfer is in progress, SIMPCOM displays a status screen that shows how much data has been transferred, elapsed time, time remaining, number of errors, efficiency rating, etc.

B+ protocol upload is supported by SIMPCOM, even though it is not listed on the menu. That is because a B+ upload must always be initiated by the host.

<PgDn>

Download a file. The process for downloading a file is essentially identical to uploading, except that if a batch protocol is in use, you are not prompted for a filename, since that information is sent by the remote. If a batch protocol is not in use, you are asked for a filename and, if the file already exists, you are asked if you want to overwrite it. You can abort the operation by answering No.

B+ protocol download is supported by SIMPCOM, even though it is not listed on the menu. That is because a B+ download must always be initiated by the host.

<AltD>

DOS shell. Invokes a second copy of COMMAND.COM, allowing you to execute other programs without leaving SIMPCOM. To return to SIMPCOM, type "EXIT" at the DOS command line. Remember that if an online session is in progress, some incoming characters may be lost if you do not return to SIMPCOM promptly.

<AltX>

Exit from SIMPCOM.

If you are writing a communication program from scratch, SIMPCOM is a better model to start from than OOPCOM, which is complex enough that its source code can be confusing when you just want an example of how to do something. Therefore examine SIMPCOM first when looking for a model to follow, and turn to OOPCOM only if SIMPCOM doesn't have what you need.

2. A Sophisticated Communications Program: OOPCOM

OOPCOM is a sophisticated communications program written with Async Professional and Object Professional. The overall look and feel of the program is similar to that of DESKPOP, the memory resident desktop manager in Object Professional. If you're familiar with DESKPOP, you'll have no trouble in learning to use OOPCOM.

Like DESKPOP, OOPCOM's display is based on the desktop metaphor, and allows multiple windows to be opened at once: a terminal window for communications, an editor window for editing text files, a file browser for viewing files while on-line, a dialer window for viewing phone books and editing phone book entries, a status window for displaying the current communications parameters, and a help window for displaying OOPCOM help text. The context-sensitive help system, activated by pressing <F1> or clicking both mouse buttons at once, can be used at any time to obtain information about the part of the program currently in use.

Although it is designed to run only in text mode, the OOPCOM user interface is similar to that of Microsoft Windows and the OS/2 Presentation Manager. All its component parts are displayed on the desktop as windows that can be moved, zoomed, and resized, and you can switch from one window to another at the press of the key. Or, if you prefer, you can use a mouse, because OOPCOM provides complete mouse support.

OOPCOM is also fully configurable. Using its companion program, OOPINST, you can change its colors and the default communications parameters—just about anything that you might want to change. This section explains how OOPCOM works and how you can use it. The following section covers OOPINST, the install program for OOPCOM.

Running the Program

To load OOPCOM, enter

```
OOPCOM [Options]
```

at the DOS command line, where Options are one or more of the following:

```
/Cfilename  Specify the name of the configuration file to use.
/M          Disable mouse support.
/?          Display list of available command line options.
```

Note that '-' can be used instead of '/' when specifying command line options.

Examples:

```
OOPCOM /M
```

Load OOPCOM with mouse support disabled.

```
OOPCOM /Cserve.cfg
```

Load OOPCOM and use the configuration settings in CSERVE.CFG.

```
OOPCOM /Cbbs.cfg
```

Load OOPCOM and use the configuration settings in BBS.CFG.

If the /C option is not used to specify a configuration file, OOPCOM tries to load the default configuration file, OOPCOM.CFG, which it looks for in the current directory, the directory that OOPCOM.EXE was found in by DOS (if you're running DOS 3.0 or higher), and all directories in the DOS PATH. If OOPCOM.CFG is not found, the default settings are used.

When it is first loaded, OOPCOM also looks for MODEM.OPL, an object oriented data file containing stream representations of all the modem objects currently implemented in Async Professional (HayesModem, CourierModem, MicrocomModem, and NullModem). If MODEM.OPL is not found, you can abort by pressing <Esc> or you can have OOPCOM use the default modem object (HayesModem).

The Main Menu

When you first load OOPCOM, your screen is divided into three parts. The top line of the screen shows OOPCOM's main menu. The bottom line is a status line. It is used to display reminders about key assignments, error messages, and prompts. Everything in between is "the desktop," the area where windows are displayed.

The main menu looks something like this:

```
≡      Files      Dialer      Terminal      Status      Help
```

Most of the choices in the main menu correspond to a window that can be opened on the desktop. To open one of the windows, simply move the highlight to the appropriate choice and press <Enter>, or click on it with the left mouse button. When multiple windows are open, you can also use the main menu to activate an inactive window.

These are your choices:

≡

About. Displays basic information about OOPCOM (copyright info, version number, etc.) in a pop-up window. Press any key to close the window.

Files

This menu choice brings up a submenu with four choices on it:

Browser

File browser. Used to view ASCII or binary files of any size.

Editor

Text editor. Used to edit text files up to 64KB in size.

Dos Shell

Temporarily drops you to DOS to run other programs.

Quit

Exit from OOPCOM.

Dialer

Phone book viewer and editor. Lets you dial a number in the phone book at the touch of a key.

Terminal

Terminal window. Displays all text sent to and received from a remote device.

Status

Status window. Displays the current communications parameters.

Help

Displays a submenu with three choices on it:

Index of Topics

Brings up an index of available help topics.

Previous Topic

Displays last selected help topic, if any.

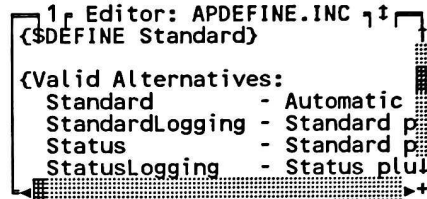
Help On Help

Displays help text concerning the help system in OOPCOM.

The main menu can be invoked from almost anywhere in the program by pressing <F10>.

Windows on the Desktop

Each of the windows on the desktop looks something like this:



```
1 Editor: APDEFINE.INC ↑↓
{ $DEFINE Standard }

{ Valid Alternatives:
  Standard      - Automatic
  StandardLogging - Standard p
  Status        - Standard p
  StatusLogging - Status plu }
```

The number at the top left corner of the border is a window number. You can use this number to jump to a particular window quickly: e.g., <Alt2> activates window #2. If you have a mouse, you can click the left mouse button on this "hot spot" to activate the window's local menu (described in the next section).

The up-and-down arrow at the top right corner is a "zoom button." Clicking the left mouse button on this hot spot zooms or unzooms a window. The Status window is the only window that cannot be zoomed.

Between these two hot spots is the window's title, which is treated as one big hot spot (or "hot region"). By clicking the left mouse button anywhere on the title, you activate a mode in which the mouse can be used to drag the window across the desktop. Click the left mouse button a second time when you are finished.

The plus sign ('+') at the bottom right corner of the border is also a hot spot. Clicking the left mouse button on it activates a mode in which the mouse can be used to adjust the size of the window. Click the left mouse button a second time when you are finished. The Status window is the only window that cannot be resized.

If the window is scrollable, it has scroll bars on the right and bottom edges of its border, as shown here. If you have a mouse installed, these can be used to jump from one place to another rapidly.

The mouse can also be used to activate an inactive window. If you can see any part of a window (other than its shadow), you can activate it immediately by clicking the left mouse button on it. Similarly, clicking the left mouse button anywhere on the top row of the screen activates OOPCOM's main menu. If the mouse cursor is on top of one of the menu choices at the time, that choice is highlighted, but not selected. (You can select it by clicking the left mouse button a second time.) If it is not on one of the menu choices, the previously selected item is highlighted, just as it is if you press <F10>.

When a new window is opened, it is placed on the desktop just on top of the previously active window. Once a window is opened, it keeps the same window number until it is closed. If a window in the middle of the sequence is closed, the next window opened is assigned its number (or the lowest available number, if more than one has been closed).

Normally, when you switch to the "next" window (using <F6>), the window with the next highest number is activated or, if there isn't one, the first window. Note, however, that the window manager

doesn't really pay attention to window numbers. If you activate an open window in the middle of the stack—using <Alt-n>, where n is its window number, or by clicking on it with the left mouse button, or by selecting it from the main menu—that window is pulled to the top of the stack. For example, if the windows were previously stacked in the order 1-2-3-4, with window 4 at the top, and you activate window 2 by pressing <Alt2>, the new order is 1-3-4-2.

Local Menus

As indicated earlier, each of the windows has a "local menu" that can be activated either by pressing <F9> or by clicking the left mouse button on the window number at the top left corner of its border. The contents of these menus vary somewhat from one window to another, but they all display at least four choices:



Close does just what its name suggests: it lets you close the current window.

Move activates a mode in which the cursor keys can be used to drag the window across the desktop to a new location. When you are finished, press <Enter>.

Similarly, Resize lets you adjust the size of the window using the cursor keys. Press <Enter> when you are finished.

Zoom lets you temporarily adjust the size of a window so that it fills the desktop. If the window is already zoomed, it is instead restored to its previous size ("unzoomed").

Resize and Zoom are disabled for the Status window because its size is fixed.

Additional choices available in a given window's local menu are described below in the appropriate section.

Hotkeys

All the functions just described can be performed using the keyboard as well as the mouse. Within all of the windows that can be placed on the desktop (except the About window), the following hotkeys can be used:

<AltM>

Move active window. Use the cursor keys to drag the window to the desired position, then press <Enter>.

<AltR>

Resize active window. Use the cursor keys to adjust the size of the window, then press <Enter>. (Does nothing in the Status window.)

<F5>, <AltZ>

Zoom/unzoom active window. (Does nothing in the Status window.)

<F6>

Activate next window. (Does nothing in Help window.)

<CtrlF6>

Activate previous window. (Does nothing in Help window.)

<Alt1>..*<Alt6>*

Activate window 1 through 6. (Does nothing in Help window.)

<F9>

Invoke local menu.

<F10>

Invoke main menu. (Does nothing in Help window.)

<AltX>

Exit from OOPCOM. (Does nothing in Help window.)

<AltS>

Show available memory.

Browser

When the Browser window is active, the status line looks something like this:

```
<F3> New file <F9> Menu <AltH> Hex/Ascii <Esc> Close | Line 1331 Col 1
```

The numbers on the right indicate the number of the line at the top of the window, and the column for the character displayed at the left edge of the window. When the Browser is reading from the disk or searching, a "Working..." message is displayed in their place.

By default, the Browser assumes that the file to be browsed is an ASCII text file. If you need to examine a binary file, press <CtrlH> or <AltH> to switch into hex mode. Pressing the key a second time switches you back to ASCII mode.

If you want to browse a different file, press <F3> and enter the name of the file. To choose the file from a directory listing, enter the appropriate file mask—e.g., C:\DOCS*.DOC—and press <Enter>.

To see a complete list of the commands that are available within the Browser window, press the help key (<F1>) or click both mouse buttons at once.

Besides the usual Close/Move/Resize/Zoom window options, the Browser's local menu offers the following choices, several of which correspond to standard keyboard commands. Where applicable, the corresponding command is shown to the right of the menu choice:

Block

Begin block <CtrlK>

Mark the beginning of a block.

End block <CtrlK><K>

Mark the end of a block.

Hidden <CtrlK><H>

Toggle the display of marked blocks.

Print <CtrlK><P>

Print the currently marked block.

Write <CtrlK><W>

Write the currently marked block to a file.

Options

Expand tabs <Ctrl>Q<T>

Toggle tab expansion (ASCII mode only). If this option is on, tabs are expanded to spaces on 8 column boundaries.

High bits <CtrlQ><H>

Toggle stripping of high bits.

Mode	<CtrlH>, <AltH>
------	-----------------

Toggle between ASCII and hex modes.

Search

Again <CtrlL>

Repeat last find operation.

Find **<CtrlQ><F>**

Search for any string of up to 30 characters.

Options

These choices let you change the default search options.

Global

Search globally?

Ignore case

Ignore case when searching?

Reverse

Search backwards?

Text markers

Jump <CtrlQ><0>..**<CtrlQ><3>**

Jump to text marker 0 through 3. You are prompted for the marker to jump to.

Set <CtrlK><0>..<CtrlK><3>

Set text marker 0 through 3. You are prompted for the marker to set.

Editor

When the Editor window is active, the status line looks something like this:

<F2> Save file <F3> New file <F9> Menu <Esc> Close | Line 24 Col 19

The numbers on the right indicate the current position of the cursor. When a search operation is in progress, the line counter is updated frequently to tell you how far the search has progressed.

To save your changes to the file being edited, press <F2>. If you want to edit a different file, press <F3> and enter the name of the file. To choose the file from a directory listing, enter the appropriate file mask—e.g., C:\DOCS*.DOC—and press <Enter>.

To see a complete list of the commands that are available within the Editor window, press the help key (<F1>) or click both mouse buttons at once.

Besides the usual Close/Move/Resize/Zoom window options, the Editor's local menu offers the following choices, several of which correspond to standard keyboard commands. Where applicable, the corresponding command is shown to the right of the menu choice:

Block

Begin block <CtrlK>
Mark the beginning of a block.

End block <CtrlK><K>
Mark the end of a block.

Change case

The choices change the case of an entire block. If no block is marked, or if the block is hidden, only the case of the character at the cursor is changed.

Lower <CtrlO><V>
Convert block to lower case.

Toggle <CtrlO><O>
Toggle the case of all characters in block.

Upper <CtrlO><U>
Convert block to upper case.

Hidden <CtrlK><H>
Toggle the display of marked blocks.

Indentation

Indent <CtrlK><I>
Indent marked block by N (indentation level) columns.

Unindent <CtrlK><U>
Unindent marked block by N (indentation level) columns.

Level <CtrlO>
Set indentation level (defaults to 2).

Print <CtrlK><P>
Print the currently marked block.

Read <CtrlK><R>
Read a file into the Editor at the position of the cursor.

Write <CtrlK><W>
Write the currently marked block to a file.

File

New <F3>
Load a new file into the Editor.

Save <F2>, <CtrlK><S>
Save the file being edited.

Write as <CtrlK><N>
Save the file being edited under a new name.

Options

Autoindent <CtrlO><I>
Toggle auto-indent mode.

Backups

If this option is on, the Editor creates a backup (BAK) file when saving a file that already exists.

Delete joins

If this option is on, pressing at the end of a line joins the current line with the following line.

Margin

<CtrlO><R>

Set the right margin.

Partial files

Load partial files? If this option is on and a new file is too large to be read in its entirety, the Editor reads in as much as possible. This setting does not affect the block read command.

Smart tabs

<CtrlO><F>

Toggle smart tabs on or off.

Tab size

<CtrlO><T>

Set tab size (fixed tabs only).

Word wrap

<CtrlO><W>

Turn word wrap on or off.

Search

Again

<CtrlL>

Repeat last find/replace operation.

Find

<CtrlQ><F>

Search for any string of up to 30 characters.

Options

These choices let you change the default find/replace options.

Block only

Search only within currently marked block?

Confirm

Confirm replacements?

Global

Search globally?

Ignore case

Ignore case when searching?

Reverse

Search backwards?

Replace

<CtrlQ><A>

Replace a string of up to 30 characters with another string.

Text markers

Hidden

<CtrlK><M>

Toggle display of text markers

Jump

<CtrlQ><0>..<<CtrlQ><3>

Jump to text marker 0 through 3. You are prompted for the marker to jump to.

Set

<CtrlK><0>..**<CtrlK><3>**

Set text marker 0 through 3. You are prompted for the marker to set.

Dos Shell

Selecting this menu option causes OOPCOM to execute a second copy of COMMAND.COM, giving you an opportunity to run other programs or execute DOS commands.

This facility is, in itself, unremarkable. What's unusual about OOPCOM's DOS shell is that it can be used even if a communications session is in progress. If you have the Terminal window open, shell to DOS, and then return to the program, the Terminal window will display the characters that were received while you were shelled out. Be aware, however, that *characters can be lost* if OOPCOM's 2KB buffer for incoming characters becomes full and the software flow control option is not enabled.

Dialer

The Dialer window is used to display and edit phone books—data files that store phone numbers and communications parameters—and to dial numbers stored in phone books for you.

When you first ask to open the Dialer window, OOPCOM tries to load the default phone book, OOPCOM.PB. If the phone book is not found, you are asked if you want to create it. If it is found, its contents are displayed in a window that looks something like this:

1 Phones: OOPCOM.PB 1	
Name	Phone number
Compuserve 2400 COS	596-0190
Compuserve 9600	1-800-331-7166
TPS BBS	260-9726
TurboPower BBS	1-719-260-9726

And the status line looks like this:

<F3> New file <F9> Menu <AltD> Dial <~N> Insert <~Y> Delete <Enter> Edit

To edit an existing phone book entry, move the highlight bar over it and press <Enter>. The data entry screen used to edit entries is the same as the one used to create them. To delete a phone book entry, move the highlight bar over it and press <CtrlY>.

To dial a phone number, move the highlight bar over the number and press <AltD>. The Dialer tries to dial the selected number using the communications parameters specified in the selected phone book entry. If a successful connection is made, the Dialer window closes itself automatically and either opens the Terminal window or makes it the current window, as appropriate.

To add a new entry to the phone book, press <CtrlN>. A data entry screen is displayed:

Download

This option downloads (receives) files sent by the remote using a file transfer protocol. When you select it, OOPCOM pops up the same window that is used when uploading. Note that it is not necessary to specify a filename unless you are using the Xmodem or ASCII protocol.

Hangup Modem

This option hangs up the modem, terminating the connection with the remote.

Options

This option brings up a data entry screen that allows you to configure OOPCOM's communications parameters and other related settings. The entry screen looks something like this:

Baud rate	2400
Parity	None
Data bits	8
Stop bits	1
Duplex	half
ANSI emulation	on
Com port	COM1
Dial prefix	
Tone or pulse	tone
Dial timeout	1092
Modem timeout	182
Delay factor	2
Words or codes	words
HW flow control	on
SW flow control	off
Modem type	HAYES

Press Ctrl-Enter to accept changes
Press Esc to abandon changes

Most of these option settings are self-explanatory. Descriptions of them can be displayed, however, by pressing the help key (<F1>) or clicking both mouse buttons at once.

Save options

This option allows you to save the current configuration settings in a file for later use. You are asked for the name of the file to store the settings in. You can either enter a filename or select an existing file to overwrite from a directory list by entering a file mask (e.g., *.CFG) and pressing <Enter>. Note that configuration files can also be created by OOPINST. See "Installation Program for OOPCOM: OOPINST" later in this section for more information.

Load options

This option allows you to restore OOPCOM to a previous state by loading configuration data from a file created with the Save options menu option. You can either enter a filename or select a file from a directory list by entering a file mask (e.g., *.CFG) and pressing <Enter>.

Status

The Status window is used to display the current communications parameters (baud rate, data bits, stop bits, parity) and the state of the hardware and software flow control options.

1 Status			
Baud:	2400	StopBits:	1
DataBits:	8	Parity:	None
Sfw Flow:	off	Hdw Flow:	on

Unlike other windows on the desktop, the Status window cannot be zoomed or resized.

Help

The Help window can be activated at any time to display context-sensitive help by pressing either <F1> or <ClickBoth>. You can also activate it by selecting the Help option from the main menu, which brings up a submenu with three options:

Index of Topics

This option opens the help window and displays an index of available help topics to select from.

Previous Topic

This option opens the help window and displays the help topic that was last selected.

Help On Help

This option opens the help window and displays information about the use of the help system.

Note that, like the About window, the Help window is "modal." That is, you cannot leave it on the screen and switch to another window. You can zoom, move, or resize it if you wish, but when you're done looking at the help text you must press <Esc> or click the right mouse button to close it.

3. Installation Program for OOPCOM: OOPINST

OOPINST allows you to customize OOPCOM to suit your personal needs and preferences. Before running OOPCOM, you should be sure that OOPCOM.EXE is in the current directory. If your intention is to establish OOPCOM's default configuration, you should be sure that OOPCOM.CFG is also in the current directory.

OOPINST expects a command line of the following form:

OOPINST [ConfigName]

The optional ConfigName parameter tells OOPINST to load the configuration data to be modified from the specified file. If you do not specify a file name, a name of OOPCOM.CFG is assumed. If the configuration file cannot be found, OOPINST uses the "factory defaults" as its starting point.

The OOPINST main menu offers four options:

Load	Customize	Save	Quit
------	-----------	------	------

Load allows you to load a new configuration file. Save allows you to save the current configuration to a file. When you select Customize, you'll see a submenu with the following options (default setting are shown to the right of each option):

Browser

Browser pages in ram	3
Number of 4KB buffers used to hold text from the file being displayed.	
Expand tabs	Yes
Expand tabs to spaces?	
Strip high bits	No
Strip high bit of each character when displaying it?	
Display in hex mode	No
Default to displaying files in hex mode?	
Default file extension	<none>
Default extension to use when prompting for filenames. (No default setting.)	

Dialer

Default phone book name	OOPCOM.PB
Filename to display as default choice when the Phone window is opened.	
Default phone book extension	PB
Default file extension for phone books.	

Editor

Editor buffer size	61440
Size of the largest file that can be loaded into the Editor. (The actual limit is 2 less than this.)	
Auto indent	Yes
Auto-indent mode on by default?	
Word wrap	Yes
Word wrap on by default?	

Delete joins lines No
 Should pressing at the end of the line cause the following line to be joined with it?
 Indent starts paragraph No
 When reformatting, should a space at beginning of a line be considered the start of a new paragraph?
 Read partial files No
 If a file is too large to be read in its entirety, should the part of it that will fit be read in?
 Make backup files Yes
 When saving a file that already exists, should the current file first be renamed to filename.BAK?
 Smart tabs Yes
 Smart tabs on by default (rather than fixed tabs)?
 Wrap at left No
 When the cursor is at column 1, should pressing <Left> move it to the end of the previous line?
 Tab delta 8
 Number of columns between tab stops (fixed tabs only).
 Default file extension <none>
 Default extension to use when prompting for filenames. (No default setting.)

Help

Default left column 3
 Leftmost column of help window.
 Default top row 15
 Top row for help window.
 Default right column 78
 Rightmost column of help window.
 Default bottom row 22
 Bottom row for help window.

Help file name OOPCOM.HLP
 Name of help file to load when program begins (change only if OOPCOM.HLP was renamed or you need to specify a complete pathname).

Modem

Drop DTR on hangup Yes
 Drop Data Terminal Ready signal when hanging up?
 Modem word or code responses words
 Configure modem to use word responses or numeric codes?
 Dial timeout 1092
 Number of clock ticks to wait for the modem to dial a number (1092 = 60 seconds).
 Command timeout 182
 Number of clock ticks to wait for the modem to respond to a non-dialing command (182 = 10 seconds).

Delay factor 2
Amount of time (in clock ticks) to delay between modem commands.

Dial prefix <none>
Any special text to be added to the modem dial command (other than 'ATD' and the phone number).

Tone or pulse dialing tone
Configure modem to use tone or pulse dialing?

Modem type HAYES
Type of modem installed (Hayes, Courier HST 9600, Microcom 9600, null modem).

Protocols

Default protocol type Xmodem
Default file transfer protocol (Xmodem, Xmodem CRC, Xmodem 1K, Xmodem 1KG, Ymodem, Ymodem G, Zmodem, Kermit, or Ascii).

Include directories No
Include directories in filenames sent to the remote?

Honor directories No
Honor directory paths specified by the remote?

Replace file mode fail
This setting determines what happens if a file about to be received already exists. If it is 'fail', the protocol will abort the transfer. If it is 'rename', the first character of the filename will be changed to '\$' (e.g., SAMPLE.DOC would be written to \$SAMPLE.DOC). If it is 'overwrite', the existing file will simply be overwritten.

Ascii protocol options

ASCII character delay 0
Amount of time to delay (in clock ticks) between characters when sending.

ASCII line delay 0
Amount of time to delay (in clock ticks) between lines when sending.

Zmodem options

Override sender No
Override sender's file management options?

Skip if no file No
Skip file if receiver doesn't already have a file of the same name.

Kermit options

Maximum packet length 80
The maximum number of bytes you want Kermit to include in one packet.

Maximum timeout 5
The maximum amount of time, in seconds, that you want Kermit to wait for a new packet or the next byte of data in a sequence.

ASCII code of pad char 0
Character used to pad front of all Kermit packets.

Pad count	0
Number of pad characters to be added to the front of all Kermit packets.	
Terminator character	13
Character used to mark the end of a Kermit packet.	
Control prefix	#
The control character prefix that Kermit uses when performing "Ctl" quoting.	
High bit prefix	Y
The prefix that Kermit uses when transforming characters with bit 7 set into 7-bit characters.	
Checksum type	1
Type of block checking Kermit is to perform: 1 = single-byte checksum, 2 = two-byte checksum, 3 = three-byte CRC.	
Repeat prefix	~
The prefix that Kermit uses when compressing long strings of repeated characters.	

Terminal

Default left column	2
Leftmost column for Terminal window.	
Default top row	3
Top row for Terminal window.	
Default right column	55
Rightmost column for Terminal window.	
Default bottom row	22
Bottom row for Terminal window.	
Scrollbar rows	200
Number of rows for scrollbar buffer.	
Default COM port	COM1
Com port to use.	
Default capture file name	OOPCOM.CAP
Default name for capture file.	
Capture buffer size	512
Size of capture buffer.	
Background buffer break	128
Maximum number of characters that TerminalWindow will buffer while it is not active before writing to the screen (16..255).	
Use ANSI	Yes
Emulate ANSI.SYS in interpretation of escape sequences?	
Half or full duplex	full
With half duplex, all characters sent to the remote are echoed to the screen. With full duplex, characters are not echoed.	
Baud rate	2400
Baud rate setting (300 baud to 115K baud).	

Parity None
 Parity setting (None, Odd, Even, Space, Mark).
 Data bits 8
 Data bits setting (5-8).
 Stop bits 1
 Stop bits setting (1 or 2).
 Software flow control No
 Enable use of software flow control (Xon/Xoff)?
 Hardware flow control
 Use DTR No
 Use DTR for receive flow control?
 Use RTS Yes
 Use RTS for receive flow control?
 Require DSR No
 Require DSR before transmitting?
 Require CTS Yes
 Require CTS before transmitting?

Colors

When you select the Colors choice from the Customize submenu, you'll see something like this:

Colors		Color Mono
Windows, General		
Text	1E 0F	
Frame	13 07	
Header, active window	3F 70	
Header, inactive window	30 07	
Shadow	08 70	
Scroll bar	13 07	
Scroll bar slider	13 0F	
Hot spots, scroll bar arrows	30 70	
Mouse cursor	4F 70	
Status line		
Text	30 07	
Edit prompt	30 07	
Edit field	31 0F	
Menus		
Frame	30 07	

The window on the right lists the various video attribute settings. Each attribute has two settings, one for color video modes and one for monochrome modes. You change a setting by moving the highlight to it and pressing <Enter>, or by clicking the left mouse button on it. The window on the left, which shows all the colors you have to choose from, then draws a box around the current setting, as shown above. Using the cursor keys, move the box to your preferred setting and press <Enter>. You can also perform this operation with the mouse. Just drag the mouse across your desk until the box is in the right place, then click the left mouse button.

When you are finished changing colors, press <Esc> to return to the main menu.

Miscellaneous

Use EMS for overlays Yes

Use EMS, if available, for the overlay file?

Use mouse if found Yes

Use a mouse, if one is present, within OOPCOM?

Use soft mouse cursor Yes

Use a software mouse cursor?

Color selection Auto

Specify how video attributes are chosen (Auto, Color, or Mono).

4. A Sophisticated Fax Program: TPFAX

TPFAX is a program that allows you to queue multiple fax files for sending to multiple recipients. In addition, TPFAX allows you to view and print faxes and convert documents into the Async Professional fax file format. It works with Class 1, Class 2, and CAS faxmodems. TPFAX expects a command line of the following form:

TPFAX [Options]

where Options are one or more of the following:

/I filename	Specify the name of the INI file to use
/P filename	Specify the name of the phonebook file to use
/Q filename	Specify the name of the outgoing fax queue file
/S	Start sending the fax queue
/?	Display help screen

Note that '-' can be used instead of '/' when specifying the command line options.

If the /I option is not used to specify an INI file, TPFAX looks for the default INI file, TPFAX.INI, in the current directory, then in the directory of TPFAX.EXE, and then in all the directories in the DOS path. If TPFAX cannot find a valid INI file, it loads with a set of default configuration options.

If the /P option is not used to specify a phonebook, TPFAX uses the phonebook specified in the INI file. If no phonebook is specified in the INI file, TPFAX uses the phonebook file TPFAX.PB. If the phonebook specified does not exist, TPFAX creates it when you add a new phonebook entry.

If the /Q option is not used to specify an outgoing fax queue file, TPFAX uses the queue file specified in the INI file. If no outgoing fax queue file is specified in the INI file, TPFAX uses the file TPFAX.OGQ. If the queue file specified does not exist, TPFAX creates it when you add entries to the queue.

The /S option tells TPFAX to send the faxes that are currently in the outgoing fax queue and then automatically exit back to DOS.

The Main Menu

When you first load TPFAX, the screen contains three things: a menu bar at the top of the screen, a desktop area where TPFAX displays its dialog boxes and status windows, and a status line that gives you short descriptions of menu items and fields in dialog boxes. TPFAX's main menu looks something like this:

≡ Convert Send View Print Phonebook eXit

Most of the menu options correspond to a dialog box where you are expected to enter information. To activate a menu choice, use the arrow keys to highlight the choice you want and press <Enter> or click on it with the left mouse button.

These are your choices:

≡

About. Displays basic information about TPFAX (copyright information, version number, etc.) in a dialog box. Press <Enter> or click on the Ok button to close the dialog.

Convert

Allows you to convert text, PCX, and TIFF files into the Async Professional fax format.

Send

Queue files for faxing. You can specify multiple recipients for the same set of files, or send just one file to one person.

View

View Async Professional fax files.

Print

Print a fax file.

Phonebook

Store the phone numbers for people you frequently send faxes to.

Exit

Exit from TPFAX to the DOS prompt.

Converting documents

Before you can send a document as a fax, you must convert it to the Async Professional fax format (APF). TPFAX can convert text files, PCX files, and TIFF files into APF format. TPFAX allows you to convert documents before sending or "on the fly" while faxes are being transmitted.

To convert documents before sending, select Convert from the TPFAX main menu. The following dialog box appears:

[.] Select Files for Conversion

Directories	Drives
- PASCAL - WINWORD - TECHFAX - WORK - TPFAX	D: ↓ Refresh

File name: _____ ↓

Available files		Selected files
- APFAX.FNT - BLTLINE.ASM - BPC.CFG - CAP.SCR - COMBOTST.PAS - CP.PAS - CVTPCL.PAS - CVTPCL.TPP - CVTPCL.TPU - DEBUG.TXT	» Select » « Remove « Ok Cancel	

At the top of the screen is a directory tree. This tree shows all of the directories on your hard disk. One of the lines in this tree is highlighted. This is the directory from which you loaded TPFAX. To change to another directory, use the arrow keys to select the directory you want and press <Enter>. If you're using a mouse, just click twice on the directory you want.

To the right of the directory tree is a drive selection field. To change drives, move to the Drives field and press <Ctrl><Down> or click on the arrow to the right of the field with the mouse. A picklist containing the names of all drives on the system appears. Choose a drive and press <Enter>.

or double click on it with the mouse. If you don't want to change drives, press the <Esc> key or click on the small box in the upper left-hand corner of the list with your mouse.

The Refresh button below the Drives field causes TPFAX to update the directory tree from your hard disk. Tab to this button and press <Enter>, or click on it with the mouse whenever you have changed your DOS subdirectory structure on the selected drive. TPFAX saves this information in a disk file to minimize startup time.

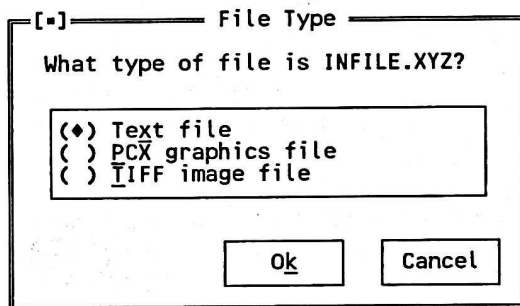
Below the directory tree is the 'File name' field. This field is used for entering the names of files that you want to convert. When you enter a file name here and press <Enter>, the name of that file is transferred to the 'Selected Files' list in the lower right-hand corner of the dialog. Up to one hundred file names can be queued for conversion.

The 'File name' field also contains a history list. This history list contains the last twenty entries that you made in the 'File name' field. You can use the history list to quickly retype previously entered items or duplicate frequently typed entries. Press <Ctrl><Down> or click on the arrow to the right of the 'File name' field to display the last twenty entries. If you select an entry in the picklist, it is placed in the file name field.

Another way to select files to be converted is to use the 'Available files' list in the lower left corner of the dialog box. Move to the available files list and use the <Up> and <Down> arrow keys to highlight the file you want to select. Press <Enter> to move the highlighted file to the 'Selected files' list. If you're using a mouse, you can highlight the file you want to select and click on the Select button.

If you need to remove a file from the 'Selected files' list, highlight the file you want to remove and press <Enter> or click on the Remove button.

When you select a file, TPFAX tries to identify the file as text, PCX, or TIFF based on its extension (TXT, PCX, or TIF). If the file has any other extension, TPFAX displays a dialog asking what kind of file it is:



Select the type of file and then the Ok button. If you need to abort the file selection, press the <Esc> key or click on the Cancel button to return to the previous dialog box without selecting the file.

After you select all of the files you want to convert, select Ok and TPFAX begins converting the documents. A convert status window is displayed:

Convert Status

Converting C:\OUTBOX\OUTFILE.TXT

Lines converted	Pages converted
22	3

The convert status window tells you which file is currently being converted and displays information about the progress of the conversion. You can abort the conversion at any time by pressing a key. A dialog box is popped up to ask if you really want to abort the conversion.

If an error occurs during the conversion, TPFAX stops converting files and returns you to the 'Select Files for Conversion' dialog box. The files that were not converted are still listed in 'Selected files' and the files that were successfully converted are no longer there. The errors that TPFAX usually encounters are input file format errors. TPFAX stops converting any time it sees something in an input file that it doesn't understand. If this happens, simply remove the offending file (it is at the top of the 'Selected files' list) and select or click on Ok to continue converting.

The converted document has the same name as the original file, with the extension APF. These converted files are stored in the same directory as the original input file. For instance, if you convert C:\DOCUMENTS\TEST123.TXT, the converted file is C:\DOCUMENTS\TEST123.APF.

Phonebook maintenance

TPFAX provides a phonebook for you to store frequently used names and fax phone numbers. When you send a fax, you can select a name and phone number from the phonebook and put that information directly in the outgoing fax record.

To access the phonebook, select the Phonebook option on the main menu. The Phonebook dialog box is displayed:

Phonebook

TurboPower Software	(719) 260-7151
Denver office	(303) 555-1212
Wichita office	(316) 555-1212

↑
↓

This dialog box contains a list of all the entries in the phonebook and buttons that you can use to manipulate the entries.

The first time you start TPFAX, the phonebook is empty. To add an entry to the phonebook, select Insert. The 'Phonebook Entry' dialog box is displayed:

[*] Phonebook

TurboPower Software	(719) 260-7151	↑ ↓
Denver office	(303) 555-1212	
Wichita office	(316) 555-1212	

[*] Phonebook Entry

Name	Phone number
TurboPower Software	(719) 260-7151

Ok Cancel

Enter the name and fax telephone number that you want to add to the Phonebook and select Ok. TPFAX returns to the Phonebook dialog and your new entry is displayed in the phonebook.

If you want to change a phonebook entry, highlight the entry and press <Enter> or click on the Change button. The 'Phonebook Entry' dialog is displayed and you can edit the entry.

If you want to delete a phonebook entry, highlight the entry and select Delete. TPFAX prompts you to confirm the deletion.

When you are finished editing the phonebook, select Ok to return to the main menu.

Sending faxes

TPFAX maintains an outgoing fax queue with information about the files to be sent, the recipients for the files, and the scheduled sending time.

To send a fax, select the Send option from the main menu. The 'Outgoing Faxes' dialog is displayed:

[*] Outgoing Faxes

Recipient	Files	Date/Time	↑ ↓
TurboPower Software	TIGER.APF, TPFMENU.APF	Immediately	
Denver office	WAKEUP.APF	11/17/93 04:00 pm	

Add to Queue Change Entry Delete Entry

Copy Send Ok Cancel

This dialog box shows you the faxes currently queued for transmission, the recipient for each fax, and when the faxes will be transmitted.

Selecting Send causes TPFAX to start transmitting the files. Selecting Ok causes TPFAX to save the fax queue to disk and return to the main menu. Selecting Cancel causes TPFAX to ignore all changes you have made to the fax queue and return to the main menu.

To add a file or list of files to the fax queue, select the 'Add to Queue' button. The 'Fax Queue Entry' dialog box is displayed:

In the upper left corner of this dialog is the 'Send files to' combo box. You can enter the name of the person or company to send the fax to, or use the arrow to the right of the field to display your list of Phonebook entries and select from the list.

To the right of the 'Send files to' field is the 'Fax number' field. Enter the fax number of the person you are sending this fax to. If you selected an entry from the phonebook, this field is filled in automatically.

The next field in the dialog box is the 'Send when' cluster of radio buttons. TPFAX gives you the option to send a file immediately or to specify the exact date and time. If you select the 'Specify date/time' radio button, you can then enter the date and time for sending the fax. When adding a new queue entry, this defaults to the date and time in your system clock.

At the bottom of the dialog is a pick list which contains the list of files to be sent to the recipient specified at the top of the dialog. By default this pick list contains no entries. To add a file to this list, select Insert. The 'Select Files for Sending' dialog box is displayed and you can select files for sending the same way you select files for conversion (described earlier in this section in "Converting Documents").

If you are using a Class 1 or Class 2 faxmodem, TPFAX tries to identify the file as text, PCX, TIFF, or APF based on its extension (TXT, PCX, TIF, or APF). If the file has any other extension, TPFAX displays a dialog asking what kind of file it is. If you specify that the file is text, PCX, or TIFF, TPFAX will convert the file to APF just before it is sent.

If you are using a CAS modem, you can send only text, PCX, or DCX files. TPFAX does not attempt to verify that the file you specify is valid.

When you finish selecting files to send, select Ok to return to the 'Outgoing Faxes' dialog box.

To change an entry in the fax queue, highlight it and select 'Change Entry'. TPFAX displays the 'Fax Queue Entry' dialog box and you can make the necessary changes.

To remove an entry from the fax queue, highlight it and select the 'Delete Entry' button.

If you want to send the same set of files to multiple recipients, you can make a copy of a queue entry. When you select the Copy button, TPFAX makes a duplicate of the currently highlighted queue entry and displays it in the 'Fax Queue Entry' dialog box. You can then change the recipient name and fax number and select Ok to add the new entry to the fax queue.

Viewing fax files

TPFAX allows you to view the Async Professional fax files that you receive or create via the conversion process. TPFAX's view features support only VGA mode \$12 (640x480 monochrome).

To view an APF file, select View from the main menu. The following dialog box is displayed:

[X] Select File for Viewing

Directories

- PASCAL
- WINWORD
- TECHFAX
- WORK
- TPFAX

Drives

D: ↓

Refresh

File name: *.APF ↓

Available files

--	--	--	--

← [Scrollbar] →

Ok Cancel

You select files for viewing in much the same way that you select files for conversion. The primary difference is that you can only view one file at a time. To select a file for viewing, type its name in the 'File name' field, choose it from the history list, or select a file from the 'Available files' list. Press <Enter> or click on the Ok button.

TPFAX puts your display adapter into graphics mode. If your adapter does not support VGA mode \$12, TPFAX reports an error and returns to the main menu.

Once the first page of the fax is displayed, you can move about using the following keys:

<Down>
Move down 1 pixel.

<Up>
Move up 1 pixel.

<Right>

Move right 64 pixels.

<Left>

Move left 64 pixels.

<PgDn>

Move down 72 pixels. Note that a complete fax page cannot usually be displayed on one screen.

<PgDn> moves down 72 pixels within the current page. To go to the next page, use <N>.

<PgUp>

Move up 72 pixels. Note that a complete fax page cannot usually be displayed on one screen.

<PgUp> moves up 72 pixels within the current page. To go to the previous page, use .

<N>

Go to the beginning of the next page.

Go to the beginning of the previous page.

<L> or <Ctrl><PgDn>

Go to the beginning of the last page.

<F> or <Ctrl><PgUp>

Go to the beginning of the first page.

<End>

Go to the end of the current page.

<Home>

Go to the top of the current page.

<Esc>

Return to the main menu.

Printing fax files

TPFAX also provides facilities for printing fax files. You can print up to 100 files at a time.

Select Print on the main menu and TPFAX displays a dialog box that is very similar to the file conversion dialog. This dialog box works exactly the same as the file conversion dialog with one important exception: only APF files can be printed. If you attempt to select a file that is not in APF format, TPFAX will not add it to the Selected files list.

Once you have selected all of the files you want to print, select Ok and TPFAX begins printing. The 'Print Status' window is displayed:

Print Status	
Printing C:\INBOX\RECEIVED.APF	
Lines printed	Pages printed
49	3

This status window tells you which file TPFAX is currently printing and the progress of that file.

If TPFAX encounters a printer error, it displays a dialog box asking you to either correct the error or abort printing. If you abort, TPFAX returns to the main menu.

If TPFAX encounters an error while printing, it stops printing and returns to the 'Select Files for Printing' dialog box. The files that were not printed are still in the 'Selected files' list.

Configuring TPFAX

TPFAX uses a Windows style INI file to store its configuration options. To change the TPFAX default options, load TPFAX.INI into a text editor (Borland's IDE, the TPE programmer's editor, OOPCOM's editor window, etc.). Each option that you can change is prefaced by a detailed comment block explaining the option and its valid values.

Following is a brief list of the options available in TPFAX.INI:

```
PhoneFile=TPFAX.PB
PrinterPort=LPT1
PrinterType=PCL
Use300DPI=True
QueueFile=TPFAX.OGQ
FaxDevice=Class2
FaxPath=
PageLength=60
Resolution=Standard
StationID=
Header=
DefCover=
ComPort=Com2
DeviceLayer=UART
ErrorCorrection=False
PortRate=9600
MaxFaxBPS=9600
RetryTime=30
DialTime=45
MaxRetries=1
ModemInit=ATS7=30M0
DialPrefix=
ToneDial=Yes
HighResConvert=True
```

For details on all of the TPFAX options, see the supplied TPFAX.INI.

G. The Help System

POPHELP is a memory-resident program designed to provide easy-to-access, online help for the Async Professional library. Since it is based on the TSR swapping capabilities of Object Professional, it occupies as little as 8KB of DOS memory. POPHELP uses help files generated by the MAKEHELP utility, also provided with Async Professional, so you can merge help files from different TurboPower products or to extend the help yourself.

If you are using the Borland Pascal 7.0 IDE, you can choose to integrate Async Professional help text directly into the IDE help system instead of using POPHELP. The following describes how to install the BP7 help file. If you are not using BP7, skip the BP7 help file discussion and read the remainder of this section to learn how to install and use POPHELP.

Installing the BP7 Help File

A Borland Pascal 7.0 IDE format help file (APRO.TPH) is provided with Async Professional. You can add it to the IDE help system as follows:

1. Start the IDE.
2. Pull down the help menu.
3. Select Files... from the help menu. The Install Help Files dialog box is displayed.
4. Select New. The Help Files dialog box is displayed.
5. Enter the path and file name for APRO.TPH (the default is \APRO\HELP\APRO.TPH).
6. Select OK to add the help file.
7. Select OK to store the help system configuration.

The Async Professional help file is now integrated into the IDE help system and available for your use.

Installing POPHELP

POPHELP can be installed in several ways:

- as a normal TSR using 150-200K of RAM
- as a TSR that swaps itself to EMS (expanded) or XMS (extended) memory, leaving only an 8K kernel behind in normal RAM
- as a TSR that swaps itself to disk, leaving an 8K kernel in RAM
- as a shell that execs another program and then automatically removes itself from memory.

You select among these options by using POPHELP command line parameters.

A help file for Async Professional is supplied with this version of POPHELP. This help file contains information about all interfaced constants, types, variables, procedures, and functions in the library. The help file is supplied in source format so that you can add to it as necessary. Compiling it is a simple one-step operation.

The INSTALL program described in §1.B stored the following files in your \APRO\HELP directory:

```
POPHELP.EXE
MAKEHELP.EXE
APRO.TXT
*.TXT
```

APRO.TXT is the main help file, which references all of the other files with extension TXT. To compile the help text, enter the following at the DOS command line:

```
MAKEHELP /Q APRO
```

Once you've created the APRO.HLP file, you can delete the *.TXT files from your disk, unless of course you intend to modify them. You can also delete MAKEHELP.EXE. This leaves two files related to help, POPHELP.EXE and APRO.HLP.

This version of POPHELP is set up to load the APRO.HLP file by default. Hence, in the simplest case, you make POPHELP memory resident by entering

POPHELP

at the command line while APRO.HLP is in the current directory. This installs POPHELP in swapping mode; it uses EMS memory (about 250K), XMS memory (about 500K, unless /I is used), or disk space (two files, unless /I is used, each about 250K) for swapping and retains about 8K of RAM (10K if /I is used).

POPHELP provides a number of options to control how it goes resident. It uses the following DOS command line syntax:

POPHELP [HelpFile] [Options]

HelpFile is an optional pathname to a help file. POPHELP automatically loads the APRO.HLP file by searching in the current directory, in the directory of POPHELP.EXE (if you're running DOS 3.0 or later), and on the DOS PATH. Specify a different directory or a different help file by putting it on the POPHELP command line. POPHELP applies a default extension of HLP to the specified file name. The following options can be specified on the command line:

/I	Single swap file (for RAM disks or XMS).
/A	Use normal file attribute (rather than hidden) for swap files.
/B	Force use of black and white video attributes.
/D	Force disk swapping.
/E ProgToExec CommandLine	Execute a program instead of going resident. No swapping. This option must be the last one on the command line.
/H Height	Specify initial help window height other than 12 rows.
/I HotKey	Specify alternate hotkey for help index.
/L HotKey	Specify alternate hotkey for screen lookup.
/M	Disable swap message.
/N	No swapping.
/P HotKey	Specify alternate hotkey for previous topic.
/S Path	Specify drive and directory for swap files.
/T	Display help text on second video monitor.
/U	Unload POPHELP.
/V	Leave help text visible on second monitor after popping down.
/X	Use XMS memory for swap.
/?	Display help screen.

Note that '-' can be used instead of '/' when specifying command line options.

When disk space is used for swapping, POPHELP normally uses two swap files. When the /I option is specified, POPHELP uses a small intermediate buffer that eliminates the need for one of the files and thus cuts its disk space requirement in half. The /I option has an analogous effect on the XMS swapping option. When /X /I is specified, POPHELP uses half as much XMS memory, swaps somewhat slower, and consumes 2K bytes more of normal RAM.

The /D option forces POPHELP to use disk space for swapping even if sufficient EMS or XMS space is available. Specify this option if you have other applications that need the EMS or XMS space, or if you have an extended memory RAM disk that you prefer to use for swapping.

When the /E option is used, POPHELP is resident only for the time the specified program is running. When the program halts, POPHELP removes itself from memory automatically. POPHELP uses COMMAND.COM to run ProgToExec, so the program can be anywhere on the DOS PATH. You can also specify any command line parameters normally used by the program, except redirection of input or output.

Since the help window is resizable after POPHELP pops up, the /H option is of minor use.

/M disables the swapping message that DESKPOP normally displays when it is swapping to disk. /M is particularly useful when you are swapping to a RAM disk.

/N tells POPHELP to run in non-swapping mode. The program uses more memory this way.

If you have a RAM disk and you want POPHELP to take advantage of it, use the /S option to give POPHELP the drive letter of the RAM disk, and it will swap to and from that area.

The /T and /V options relate to using two monitors with POPHELP, one for the source code and one for the help text. If /T is used, POPHELP displays its output on the monitor that was not active when POPHELP went resident. If /V is used, help text is left visible on the second display after popping down.

The /X option tells POPHELP to use XMS memory for swapping even if EMS is available. If /X is not specified and both EMS and XMS are present, preference is given to EMS.

The following command lines provide common examples of using the options.

```
POPHelp C:\APRO\APRO
```

Load APRO.HLP from the specified directory.

```
POPHelp /D /M /S F:\ /1
```

Force POPHELP to swap to disk, using the root directory of drive F: for a single swap file. Disable swap messages. The single swap file is most often appropriate on RAM disks, which are very fast but have relatively limited storage space.

```
POPHelp /X /1
```

Enable use of XMS (extended) memory. Note that in order to use XMS you must load an XMS-compatible driver such as HIMEM.SYS, 386MAX.SYS, or QEMM386.SYS. The /1 option causes POPHELP to use half as much extended memory space, with slight degradation in performance.

```
POPHelp /E TURBO MYPROG.PAS
```

Load POPHELP temporarily and run the TURBO integrated environment.

```
POPHelp /N
```

Make POPHELP resident without swapping (keeps lots of DOS memory).

```
POPHelp /U
```

Unload the resident copy of POPHELP.

```
POPHelp /B
```

Force POPHELP to use only black and white (\$07, \$0F, \$70) video attributes.

Display help text on the second video adapter (the one that is currently inactive). When POPHELP exits, the help text will remain visible on the second display.

Keep in mind one important restriction when installing POPHELP in swapping mode. When POPHELP pops up, it disables any TSRs loaded chronologically after it. This is necessary because it probably overwrites their memory while it swaps in. Some TSRs, especially network shells and communication programs, won't accept being disabled in this way: the network will disconnect a station that doesn't respond after a time, or a communication program will lose incoming characters. It's possible to avoid these problems by loading POPHELP after such TSRs. To avoid one common problem, POPHELP automatically detects when it is loaded before the Novell NetWare shell and refuses to pop up in this situation.

Hotkeys

POPHELP uses three different hotkeys. By default, the hotkeys are:

<LeftShift><F1>

Look up a help topic based on the word at the cursor

<LeftShift><F2>

Redisplay the previous help topic

<LeftShift><F3>

Display the help index

The following options control the hotkeys:

/L Specify alternate hotkey for screen lookup
 /P Specify alternate hotkey for previous topic
 /I Specify alternate hotkey for help index

You can specify alternate hotkeys when POPHELP is installed. On the command line, a hotkey is specified as a hexadecimal word. The top byte specifies the shift key(s) to be pressed and may be zero. The bottom byte specifies the scan code for the hotkey.

The shift key codes are:

No shifts - 00
 RightShift - 01
 LeftShift - 02
 Ctrl - 04
 Alt - 08

Valid scan codes (in hexadecimal) are:

A - 1E	N - 31	0 - 0B	F1 - 3B	[- 1A
B - 30	O - 18	1 - 02	F2 - 3C	; - 27
C - 2E	P - 19	2 - 03	F3 - 3D	, - 33
D - 20	Q - 10	3 - 04	F4 - 3E	/ - 35
E - 12	R - 13	4 - 05	F5 - 3F	\ - 2B
F - 21	S - 1F	5 - 06	F6 - 40	] - 1B
G - 22	T - 14	6 - 07	F7 - 41	' - 28
H - 23	U - 16	7 - 08	F8 - 42	. - 34
I - 17	V - 2F	8 - 09	F9 - 43	` - 29
J - 24	W - 11	9 - 0A	F10 - 44	
K - 25	X - 2D		F11 - 57	
L - 26	Y - 15		F12 - 58	
M - 32	Z - 2C			

For example:

/L 0244	performs lookup when <LeftShift><F10> is pressed
/P 0819	shows previous topic when <Alt><P> is pressed
/I 0517	displays help index when <Ctrl><RightShift><I> is pressed

Two hotkeys may not be based on the same scan code, even if the shift keys differ. For example, you cannot use <Alt><F1> for one hotkey and <Ctrl><F1> for another.

Using POPHELP

Once POPHELP is installed, you can go about your business. The most likely place for you to start using it is within your text editor or within the TURBO integrated environment.

If you want to use a particular Async Professional routine and you can remember its name but not its parameters, position the editor cursor within or immediately after the procedure name you typed. Then press the Lookup hotkey, <LeftShift><F1> by default. POPHELP searches the list of topics looking for an exact match (disregarding case). If that fails, it looks for a topic whose name starts with the search string.

If POPHELP can find a matching topic with one of these two search strategies, it pops up a help window showing you the details about that routine. If it can't find a match, it pops up a window with the names of all the Async Professional units.

If you're not sure of the name of a routine, but you remember which unit it's in, press the Index hotkey, <LeftShift><F3> by default, and you'll get a pick list showing all the Async Professional units. Select the unit you want and a list of topics within that unit appears.

In the help window, you can use the arrow keys to move the cursor to highlighted areas that indicate links to other topics. Once the cursor is positioned in a highlighted region of interest, press <Enter> to bring up that topic. This allows you to browse through the help database. The <Tab> and <ShiftTab> keys make the cursor jump immediately from link to link.

Once you've seen the information you want, you can return to the underlying application by pressing <Esc>. If you want to see the same help topic again, press the Previous Topic hotkey, <LeftShift><F2> by default.

Pasting Help Text into your Editor

Among its other capabilities, POPHELP can also paste marked blocks of help text into your editor. Cut/paste works much the same way that block copying works in the TURBO integrated environment.

The first step is to position your editor's cursor at the exact location where you want to paste the help text. Next, pop up the help screen and select your topic. Move the cursor through the text until it is positioned on the first character that you want to paste. Mark the beginning of the block by pressing <F7> (or <CtrlK>). Scroll through the text until the cursor is positioned just beyond the last character that you want to paste. Press <F8> (or <CtrlK><K>) to mark the end of the block. POPHELP highlights the block. Finally, press <F4> (or <CtrlK><C>) to copy the marked block into your editor. POPHELP immediately pops down and starts feeding keystrokes to the underlying application. The next time you invoke POPHELP, the block is hidden. If you want to toggle the block's visibility, press <CtrlK><H>.

If the pasted text causes your editor to scroll, the display may temporarily look strange since POPHELP can type much faster than the editor was probably designed to accept. However, your editor will "catch up" when POPHELP finishes pasting characters.

If your editor has an automatic indentation feature, you will probably want to disable it—e.g., `<CtrlQ><I>` toggles autoindent in the TURBO IDE—during the paste. If autoindent is on during pasting, the pasted text may gradually move to the right of the screen.

If the pasted block is larger than POPHELP's internal paste buffer (512 keystrokes), POPHELP must swap itself into memory to refill the buffer each time it is exhausted. When this occurs, the pasting is temporarily delayed until the buffer is refilled and POPHELP swaps out again. Depending on the speed of your system and whether POPHELP swaps to EMS, XMS or disk, this delay can range from a fraction of a second to a few seconds. If POPHELP cannot safely swap itself in when the buffer empties, the paste is aborted prematurely. This should not occur when POPHELP is used over modern programmer's editors; EDLIN may have trouble.

POPHelp Command Summary

The following table summarizes the POPHELP commands.

`<F1>`

Display the master topic index.

`<AltF1>`

Display the topic that was displayed just prior to the current one. The cursor is positioned to its last location in the help window.

`<F2>`, `<CtrlQ>`, `<F>`

Search for a text string. If a non-empty string is entered, POPHELP searches the complete help text for the string. Searching is case-insensitive. POPHELP searches from the position of the cursor until it finds a match or reaches the current topic again. If a match is found, the cursor is moved to the beginning of the match.

`<F3>`

Prompt for a new help file. A default extension of HLP is added to the name you enter. POPHELP reads any help file compiled using the MAKEHELP utility provided with Async Professional or Object Professional. Note that each help file needs memory space that is proportional to the help file's size and number of topics. POPHELP reserves a certain amount of memory when it goes resident and it cannot increase that amount later. Therefore, if you expect to load multiple help files during one POPHELP session, you should load the largest of them when POPHELP first goes resident.

`<F4>`, `<CtrlK>`, `<C>`

If a block is marked and visible, paste it from the help system to the underlying editor.

`<F5>`, `<AltZ>`

Toggle the zoom mode of the window.

`<F6>`, `<AltM>`

Enter move/resize mode. Use the cursor keys to move the window around the screen. This moves the window in small jumps rather than one row or column at a time. Use the Shift-cursor keys to resize the window one row or column at a time. You can also use "speed resizing" to resize the window in small jumps. Use `<Ctrl>``<LeftArrow>` and `<Ctrl>``<RightArrow>` for horizontal speed resizing; `<PageUp>` and `<PageDown>` for vertical speed resizing. The window cannot be resized while it is zoomed.

`<F7>`, `<CtrlK>`, `<B>`

Mark the beginning of a block.

<F8>, <CtrlK><K>

Mark the end of a block.

<F9>, <CtrlL>

Perform the last search again. If the last search performed was a topic name search (activated by a <ShiftF1> popup), this command repeats the topic name search. Searching starts with the next topic beyond the current one and is circular, wrapping from the last topic to topic 1 as many times as you like. This feature is especially useful when you're searching for a member function that has a common beginning, such as "Get".

If the last search performed was a text search (activated by <F2> or <CtrlQ><F>), this command repeats the text search. Searching starts just after the last match. Previous matches are stored on the topic stack and can be reviewed by pressing <AltF1>. A subsequent <F9>, however, continues from the position of the last match rather than from the current screen.

<CtrlK><H>

Hide or unhide the currently marked block. <F4> pastes only if a block is visible.

<PgUp>, <PgDn>

Move to the next or previous page of the current help topic.

<Left>, <Right>, <Up>, <Down>

Move the cursor around the help screen. Attempting to move the cursor beyond the edge of the window causes more text to scroll into the window.

<Tab>

Jump to the next cross reference topic, if any.

<ShiftTab>

Jump to the previous cross reference topic, if any.

<Enter>

Select the cross-reference topic at the cursor location, if any. The new topic is loaded into the window and the cursor is positioned on the topic's first character.

<Esc>, <AltX>

Exit POPHELP. The current topic and page are retained and can be redisplayed by pressing the Previous Topic hot key, <LeftShift><F2> by default.

H. Calling for Technical Support

TurboPower Software offers telephone support on an as-available basis at no extra charge from 9 a.m. to 5 p.m. Mountain time. A CompuServe forum area is also maintained to support you. The account and telephone numbers are listed on the title page of this manual.

Technical support is always a tough job and throwing communications problems into the equation makes the task even tougher. For that reason, we ask you to do several things before calling for support. These may seem like trivial things (and some of them are indeed trivial) but getting them out of the way ahead of time could save you the cost of a phone call.

First and foremost, if you're writing an application and "not getting anything" please try the supplied, unmodified, precompiled demonstration programs COMTEST and OOPCOM. This is a polite way of saying "make sure it's plugged in" before deciding your application doesn't work. Whether you're connecting to a piece of data collection equipment, plugging in a new plotter, or just trying to send commands to a modem, start from a known, reliable program to prove to yourself that the device is hooked up, properly configured, and connected with a working cable.

If you've proven that all is well with your hardware but your program still isn't behaving properly, be sure to use some of the Async Professional built-in debugging tools to try to find the problem. Tracing is particularly handy for tracking down failed protocols and scripts gone awry. We recommend using the "Tracing" conditional define to turn Tracing on and off in your programs. That is, if your program looks something like this:

```
begin {main block}
  {$IFDEF Tracing}
    InitTracing(5000);
  {$ENDIF}

  {...rest of program...}

  {$IFDEF Tracing}
    DumpTrace('EXAMPLE.TRC');
  {$ENDIF}
end.
```

Then you can turn Tracing on and off by simply enabling or disabling the {\$DEFINE Tracing} directive in APDEFINE.INC. Of course, your program must include APDEFINE.INC for this to work. Be sure to read Chapter 4 for the complete details on how to use the Tracing facility in your program.

EventLogging is useful for figuring out flow control issues and for proving to yourself that your program really did receive (or send) the proper data. As with Tracing, you can control EventLogging in your program with the {\$DEFINE EventLogging} directive in APDEFINE.INC.

Finally, Async Professional provides a user error procedure for custom handling of errors. We strongly recommend that every program use such a procedure. A fair percentage of technical support calls are the result of an application program continuing to use an object (or unit) after an error has been reported. In the case of Async Professional, you can either check AsyncStatus after every library call or you can install your own error handler and pop up a window whenever it receives an error. Please read "User Error Procedure" in §4.B for more information.

If you tried a "known good program" and applied all the built-in debugging tools and you're still having a problem figuring out what's going on, then give us a call or leave a message in our area of CompuServe and we'll do our best to help you find a solution.

Depending on the problem you're having, we may ask such questions as "What did COMTEST do in that situation?" or "Did you try FX or SIMPCOM?" or "What error number was reported?". If you have answers to such questions handy, we'll probably be able to zero in on the problem much faster. We might also need to discuss your trace file or event log file. Please be sure to have such files available when the problem warrants it.

Note that these suggestions cover only problems with the communications aspects of your program. We're also available to help with the more mundane issues of debugging: using a routine incorrectly, failing to close a port, memory overwrites, and so on. We're used to answering such questions and we'll continue to do so for Async Professional. However, many times the problems we hear on the technical support line don't really have anything to do with our libraries but are just general debugging problems that programmers deal with every day.

For more handy tips on debugging communications programs, be sure to read Appendix C.

I. Purchase Agreement

This software and accompanying documentation are protected by United States copyright law and also by International Treaty provisions. Any use of this software in violation of copyright law or the terms of this agreement will be prosecuted to the best of our ability.

Async Professional is copyright (c) 1991 and 1993 by TurboPower Software. Portions are copyright (c) 1987-1989 by Information Technology Ltd., and are used under license to TurboPower Software.

TurboPower Software authorizes you to make archival copies of this software for the sole purpose of back-up and protecting your investment from loss. Under no circumstances may you copy this software or documentation for the purposes of distribution to others. Under no conditions may you remove the copyright notices made part of the software or documentation.

You may distribute, without runtime fees or further licenses, your own compiled programs based on any of the source code of Async Professional. You may also distribute any of the DynaLink Libraries (DLL files) provided with Async Professional. You may not distribute any of the Async Professional source code, compiled units, or compiled example programs without written permission from TurboPower Software.

Note that the previous restrictions do not prohibit you from distributing your own source code or units that depend upon Async Professional. However, others who receive your source code or units need their own copies of Async Professional in order to compile the source code or write programs that use your units.

The supplied software may be used by one person on as many computer systems as that person uses. We expect that group programming projects making use of this software will purchase an individual copy of the software and documentation for each member of the group. Contact TurboPower Software for volume discounts and site licensing agreements.

With respect to the physical diskettes and documentation provided with Async Professional, TurboPower Software warrants the same to be free of defects in materials and workmanship for a period of 30 days from the date of receipt. If you notify us of such a defect within the warranty period, TurboPower Software will replace the defective diskette(s) or documentation at no cost to you.

TurboPower Software warrants that the software will function as described in this documentation for a period of 30 days from receipt. If you encounter a bug or deficiency, we will require a problem report detailed enough to allow us to find and fix the problem. If you properly notify us of such a software problem within the warranty period, TurboPower Software will update the defective software at no cost to you.

TurboPower Software further warrants that the purchaser will remain fully satisfied with the product for a period of 30 days from receipt. If you are dissatisfied for any reason, and TurboPower Software cannot correct the problem, contact the party from whom the software was purchased for a return authorization. If you purchased the product directly from TurboPower Software, we will refund the full purchase price of the software (not including shipping costs) upon receipt of the original program diskette(s) and documentation in undamaged condition. TurboPower Software honors returns from authorized dealers, but cannot offer direct refunds to anyone who did not purchase a product directly from us.

TURBOPOWER SOFTWARE DOES NOT ASSUME ANY LIABILITY FOR THE USE OF ASYNC PROFESSIONAL BEYOND THE ORIGINAL PURCHASE PRICE OF THE

SOFTWARE. IN NO EVENT WILL TURBOPOWER SOFTWARE BE LIABLE TO YOU FOR ADDITIONAL DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THESE PROGRAMS, EVEN IF TURBOPOWER SOFTWARE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

By using this software, you agree to the terms of this section. If you do not agree, you should immediately return the entire Async Professional package for a refund.

TurboPower Software provides CompuServe and telephone support for Async Professional on an as-available basis at no extra charge. We encourage you to use CompuServe for support questions. When you call for technical support, the person who happens to answer the phone will try to answer your question on the spot. When you ask a question on CompuServe, the person most qualified to answer your question will respond. CompuServe access information and the technical support telephone numbers are listed on the title page of this manual.

2. Principles of Operation

A. Overview

This chapter covers two topics: serial communications in general and the architecture of Async Professional.

If you are already familiar with the concepts and facilities of serial communications, you might want to skip that section. However, you will at least want to skim the section on the architecture of Async Professional. It's a bit different from communication libraries you might have used in the past and a few minutes spent here can save you a lot of questions later.

B. Basics of Asynchronous Serial Communication

Rather than present an exhaustive description of all the myriad details of asynchronous serial communications, this section presents those concepts and facilities that you need to understand to use Async Professional. These are just the basic concepts (line parameters, line errors, and so on) that you need to understand before you can even hook a serial device to your PC.

After presenting these basic concepts, this discussion continues into some of the lower-level details of serial communications (starting with a detailed discussion of the UART chip). Generally, you do *not* need such detailed knowledge to use Async Professional.

The broadest definition of serial communications includes anything that transmits or receives data in a serial fashion—where one bit follows another in a single stream over one wire. In parallel communications, by contrast, many bits are sent in parallel over many wires. Asynchronous serial communications simply means that the data stream includes start and stop bits, bits that mark the beginning and end of each character in the data stream. This is in contrast to synchronous communications where no start or stop bits are provided and the two ends of the link rely on synchronized clocks to know where each character starts and stops.

In the PC world, when someone speaks of serial communications they are invariably talking about the communications facility provided by the "serial ports" (or "com ports") at the back of a PC. To these ports we attach a wide variety of peripheral devices. We attach modems so that we can call other computers. We attach printers and plotters. We attach input devices like data acquisition equipment and laboratory instruments. In fact, we can attach and communicate with anything that adheres to the same serial communications standard as the serial port.

Let's talk for a moment about some of the terminology used in this manual. Given the wide variety of serial peripherals that someone might be using and the corresponding variety of applications, it's sometimes hard to find a general purpose descriptor for what you have attached to your serial port. Should we call it a modem? printer? instrument? remote device? attached device? that thing at the other end of the line? When the term is too general it can water down the point we want to make. When the term is too specific it might not be clear how the point relates to other cases. So, we try to use the appropriate terms as they are called for. We say "device" or "remote device" (or something like that) when it's not really important what you are working with. If the point we are trying to make *is* related to the attached device, then we try to be more specific about what that device is.

In subsequent discussions we talk about something called a UART (Universal Asynchronous Receiver/Transmitter). This is the chip within your PC that handles the low-level details of receiving and transmitting data. You don't really need to know any more beyond that. However, we will occasionally use this term and you should know what we are talking about. If you *are* interested in learning more about the UART, read "Universal Asynchronous Receiver/Transmitter (UART)" later in this section.

Line Parameters

With serial communications you don't need to know all of the nitty gritty details about the line voltages and pin names and so forth. You do need to know, however, about the line parameters: baud rate, parity, data bits and stop bits. Usually, such things are expressed like this:

9600,N,8,1

This describes a communications link operating at 9600 baud, no parity bit, eight data bits, and one stop bit. Before any communication can take place, both ends of the link must be using the same line parameters.

You specify line parameters in your Async Professional application whenever you open a serial port. In some cases, you might not have any leeway at all—the device you want to communicate with might force some particular line parameters upon you. More likely, though, you'll need to choose appropriate line parameters for your PC and the device attached to your serial port. Here are some brief definitions of these line parameters and some guidelines for choosing appropriate values.

Baud Rate

Baud rate is commonly used to mean bit rate—the number of bits transmitted per second. This is technically incorrect. Baud rate actually means the number of events per second in a communications line. Since an event can contain information about more than one bit, as happens with high-speed modems, baud rate could be quite different than bit rate. At the serial port itself (which is what Async Professional usually concerns itself with) each event is a single bit, so equating baud rate and bit rate works out OK.

When given a choice, you should generally select a baud rate as high as possible to give you the highest possible throughput. Understand, however, that there are very likely other limits in your application or environment that might limit your throughput. The first is the speed of your PC. Here are some rules of thumb for highest reasonable baud rate on various machines:

4.77 Mhz PC (8088)	9600 baud
8 Mhz PC (8088)	19200 baud
8 Mhz AT (286)	57600 baud
16 Mhz AT (386)	115200 baud
386 or 486 with a memory manager	57600 baud

Note that these are rules of thumb only. Depending on what your application is doing, you might get higher or lower throughput. Also, the higher the baud rate, the more likely you are to get transmission errors (line errors). If you are getting a lot of line errors at 38400 baud (resulting in a lot of retransmitted data), you might actually get an increase in throughput by dropping down to 19200 baud since less data will need to be retransmitted due to errors.

Other factors affect your choice for baud rate as well. For example, if you're using a 2400 baud modem there is little reason for selecting 38400 baud between your PC and that modem. Or, if you are collecting data from some device that sends only a few hundred bytes of data per minute, there's no sense in selecting a high baud rate. You would do better to select a lower baud rate and minimize transmission errors.

Data Bits

A data byte can contain 5, 6, 7 or 8 bits. The vast majority of applications will use either 7 or 8 bits since most data is expressed in 8-bit bytes (though text data can often be expressed in only 7 bits).

Many time-sharing systems, such as Compuserve, work with only 7 data bits because that's all they need to display text data. When transferring binary data though, with a file transfer protocol for example, you almost always switch to 8 data bits. In fact, Kermit is the only protocol in Async Professional that will work with 7 data bits.

Stop Bits

Stop bits follow the data bits in the serial data stream. The value for stop bits is always either 1 or 2. Generally, you'll use 1 stop bit.

Parity

Parity describes a bit checking scheme. When used, all of the bits in a data byte are added together. A final bit, called the parity bit, is added such that the sum of all bits is either odd or even (whichever you specify). The transmitter calculates and transmits a parity bit. The receiver also

calculates a parity bit and compares it to the parity bit it received. If the bits are equal, then it is assumed that the character was received without error. If the bits are not equal, then it is assumed that there was an error during transmission.

The possible values for parity are:

- NoParity - no parity bit is added
- EvenParity - a parity bit is added such that the bit sum is always even
- OddParity - a parity bit is added such that the bit sum is always odd
- MarkParity - a parity bit of value one is always added
- SpaceParity - a parity bit of value zero is always added

Whether or not you should use parity depends on your application. Generally, you wouldn't need to use parity bits if your application relies on some other means of checking data integrity (like block check characters).

Line Errors and Breaks

Serial I/O, like all forms of I/O, is subject to errors. An error exists when the character(s) received by the receiver are different from what the transmitter sent. This usually happens when the data line is disturbed by some electrical interference. All communications links, serial or otherwise, are subject to such transmission errors. Your programs *must* be prepared to deal with these errors. Typical actions are to discard and ignore such data or to ask the transmitter to send that data again. The action you take depends on the requirements of your application.

Line errors can also occur if the receiver and transmitter are using slightly different line parameters. And, finally, line errors can occur if you've selected a baud rate that is just too high for the environment in which your application is running.

Here are the types of line errors that can occur, what they mean, and how you might deal with them:

UART Overrun Error

This error means that a second character arrived at the serial port before the first one was processed. Generally, this means that you are running at a baud rate that is too high and your machine is not fast enough to handle the characters as fast as they are arriving. The usual solution is to lower the baud rate until the UART overrun errors go away.

In some cases the problem is not that the baud rate is too high, but that some other process going on in your machine is leaving interrupts inhibited for too long. All incoming characters generate an interrupt that tells Async Professional to read that character. If your application (or some TSR or device driver) has turned interrupts off, then Async Professional isn't notified that a character arrived in time to read it.

So, if you get UART overrun errors and you believe that your machine should be able to handle the current baud rate, check your application and operating environment for code that might be leaving interrupts off. Here are some situations that might not be immediately obvious.

Snow-checking algorithms for CGA monitors turn interrupts off while waiting for horizontal or vertical retraces (the FastWrite routines in Turbo Professional and Object Professional do this). Usually, interrupts are off for a very brief period, but it could cause problems above 19200 baud.

Some EMS boards/drivers are fairly slow when mapping logical pages into physical pages. We've heard that some can leave interrupts off for many milliseconds. Our experience hasn't been quite this bad, but we have seen the problem. For us, it occurred while testing a program whose overlays were stored in EMS. If the EMS manager is called too frequently to remap pages, which can happen if overlay thrashing occurs, the activity can lead to UART overruns.

Some network shells (for example, Novell's NETX) leave interrupts disabled while writing to network drives. This is most apparent when downloading files (via ProtocolReceive) where the destination drive is a network drive. When the protocol needs to write data to the disk, the network shell often leaves interrupts off long enough to cause UART overruns. The solution in this case is to turn on the apRTSLowForWrite option discussed in §7.C.

The hardware keyboard interrupt (int 9), which is called whenever a key is pressed or released, often leaves interrupts off from 200 microseconds to a full millisecond. This can cause UART overruns at speeds as low as 9600 baud. There are a wide variety of interrupt times from machine to machine. In practice, this rarely proves to be a problem since the operator of a communications program usually isn't typing at the keyboard when data is streaming in at a high rate. If it does prove to be a problem in your application, you have three alternatives: (1) use a lower baud rate; or (2) take over interrupt 9 and lower RTS while the interrupt is being serviced (this requires that hardware flow control is turned on at the remote); or (3) use RotateIrqPriority to give the serial port a higher interrupt priority than Int 9 (see RotateIrqPriority in §4.B).

If you are using protected mode, the additional overhead can cause UART overruns at a lower baud rate than an equivalent real mode program. See §2.E for more information on using protected mode.

Parity Errors

Parity errors occur when the parity bit received differs from the parity bit calculated. For example, if the receiver specified OddParity and receives a character with EvenParity, that is a parity error. By far the most common cause of parity errors is a mismatch in the parity line parameter between the transmitter and the receiver. Always suspect this if you get a lot of parity errors when you first connect to a new device. Another clue is when certain characters always return parity errors and certain other characters never return parity errors.

Parity errors can also be caused by interference on the data line (i.e., transmission errors). If this is the case, errors occur randomly with groups of errors interspersed with long periods of error-free transmission. In this case about the only solution is to try rerouting the cable (away from any sources of electrical interference). Shortening the cable run could also help.

Framing Errors

A framing error occurs when the data bits in the serial stream are not followed by a valid stop bit (a stop bit must always have the value '1'). As with parity errors, the most likely cause of framing errors is a mismatch in line parameters between the receiver and transmitter. Always verify the line parameters at both ends of the communication link whenever you get framing errors.

Framing errors can also be due to electrical interference with the serial cable (true transmission errors). Again, about the only options available for fixing such errors are shortening or rerouting the cable run.

Breaks

Breaks are not really line errors, but they do represent a special line condition. A "line break" (or just "break") is a condition in which the data line is filled with zero bits for as long as it takes to send one character (one "character time"). The UART recognizes this special condition and can notify you that it has received a break. Breaks are used when one device needs to signal, or interrupt, another device to have it handle some special condition. Breaks aren't used much these days, however, Async Professional can both send and detect breaks.

Universal Asynchronous Receiver/Transmitter (UART)

Generally, you don't need to understand the specifics of UART operation to use Async Professional. In some circumstances, though, you might find such background information helpful for thinking through some debugging problem; or, you just might want a clearer picture of what Async Professional is really doing. If you're just getting started with Async Professional, or you already know all you care to know about UARTs, you might want to skip this section.

The brain of the serial communications facilities on IBM PCs, PS/2s, and compatibles is a chip called a Universal Asynchronous Receiver/Transmitter (UART). In nearly all cases this chip is from a family of National Semiconductor integrated circuits. Older PCs use UARTs with chip designations INS8250 and INS8250B. Newer machines use NS16450 and NS16550 chips. Although there are slight differences between the chips in speed, internal behavior, and features, their basic properties are the same, and Async Professional works equally well with any of them.

The UART is responsible for all of the grunt work of serial communications. It transmits data by taking a byte and serializing the bits onto the output line. It receives data by reading a stream of bits from the input line and de-serializing them into a data byte. The UART also controls the line parameters discussed earlier (baud rate, data bits, stop bits and parity). And the UART is responsible for setting and reacting to various line and modem control and status signals.

The UART does all of these things in response to requests from an application program. The program "talks" to the UART through the UART's registers. To the program, these registers are nothing more than addresses somewhere in the PC's I/O address space (accessed by Turbo Pascal's built-in Port and PortW arrays).

The IBM PC architecture also associates a hardware interrupt with each UART. You can use the serial port without using the interrupt, but it's generally not practical to do so. (The APUART device layer of Async Professional takes complete advantage of the interrupt services available.) The names that are typically used to refer to serial ports (Com1, Com2, etc.) tell us the address of the UARTs and what hardware interrupt they use.

These addresses and interrupts are governed by two standards: the IBM PC standard for Com1 and Com2 and the de facto standard for Com 3 and Com4 on IBM PCs and compatibles; and the IBM PS/2 standard for Com1 through Com8. Here are those standards:

IBM PC Standard (Com1, 2) and de facto Standard (Com3, 4)

ComName	base address	IRQ	Vector
Com1	0x03F8	4	0x0C
Com2	0x02F8	3	0x0B
Com3	0x03E8	4	0x0C
Com4	0x02E8	3	0x0B

IBM PS/2 Standard

ComName	base address	IRQ	Vector
Com1	0x03F8	4	0x0C
Com2	0x02F8	3	0x0B
Com3	0x3220	3	0x0B
Com4	0x3228	3	0x0B
Com5	0x4220	3	0x0B
Com6	0x4228	3	0x0B
Com7	0x5220	3	0x0B
Com8	0x5228	3	0x0B

When opening serial ports in Async Professional, you always identify the port by its ComName (Com1..Com8). This is just a shorthand way of specifying a UART base address and an interrupt vector. In some cases you may have a UART that uses a non-standard base address and/or interrupt vector. Async Professional can handle this without difficulty, but you'll need to use the SetUart routine to specify the base address and interrupt vector you want to use.

Unfortunately, a conflict arises between each standard's definition of Com3 and Com4. Async Professional tries to resolve this conflict internally. That is, when you ask to open Com3 or Com4, Async Professional determines whether you are running on a PC/AT or PS/2 and uses the appropriate base address and interrupt vector. (See PS2DetectMode in the APPORT module in §4.B for more information.)

Using Multiple Ports

Since the standard PC bus dedicates only two interrupt lines (IRQs) to COM ports, you can usually only open two ports simultaneously. However, some serial port hardware allows other IRQs to be selected for a COM port. If by doing this you can assure that all open COM ports will have different IRQs, then you can open up to four ports simultaneously. If you select non-standard IRQs, you must inform Async Professional as follows:

```
SetUart(Com3, $3E8, 5, 0);
SetUart(Com4, $2E8, 7, 0);
```

These calls *must* be made before the COM port is opened (whether using procedural or OOP syntax). These calls simply inform Async Professional that you made non-standard assignments. Your hardware must be capable of supporting such assignments.

Registers

Async Professional communicates with a UART via the UART registers. It controls the UART by writing information into its registers and it retrieves data and status information by reading these registers. A UART contains eight registers, each with a specific purpose, starting at the base address and continuing for the next eight port addresses. The register at the base address is called register 0, the next register is register 1, and so on. Since the Com1 UART's base address is \$3F8, we know that register 0 is also at \$3F8, register 1 is at \$3F9, and so on up to register 8 at \$3FF.

Each of these registers also has a name. Registers that provide status information when read are designated (read). Registers that are used to program the UART are designated (write). Those that are used both ways are designated (read/write).

Register 0: receiver buffer register (read)
 transmit holding register (write)
 divisor latch low (read/write)

data bit7	data bit6	data bit5	data bit4	data bit3	data bit2	data bit1	data bit0
7	6	5	4	3	2	1	0

Register 0 bit definitions

Register 0 has three names and three purposes. When you read from register 0, you are reading the latest received character (assuming there is a received character). When you write to register 0, you are passing the next character to be transmitted (assuming the UART is ready to transmit a character).

The third purpose of register 0 comes into play when setting the baud rate. When the divisor latch access bit is set, register 0 specifies the low byte of the baud rate divisor. The baud rate divisor is a value which, when divided into a preset constant, yields the desired baud rate. This "preset constant" is determined by the internal clock rate of the machine, which is the same for all PCs.

The baud rate divisor is determined by this equation:

$$\text{divisor} = 115200 / \text{baud rate}$$

Hence, the process for setting the baud rate on a UART is to first calculate the baud rate divisor, then set the divisor latch access bit, write the low byte of the divisor to register 0, write the high byte of the divisor to register 1 and finally, clear the divisor latch access bit.

Register 1: **interrupt enable register (write)**
 divisor latch high (read/write)

N/A	N/A	N/A	N/A	modem status	line error/break	xmit ready	recv char
7	6	5	4	3	2	1	0

Interrupt enable bit definitions

UARTs can generate an interrupt in response to four different conditions. Programs specify which interrupt conditions they want to enable by writing into this register. Here are the four interrupt conditions:

1. A character was received
2. The transmitter just finished transmitting a character
3. An error or break signal occurred
4. A modem status signal changed

To enable a particular interrupt, set the proper bit (see the bit definitions above) in a byte mask and write the byte mask to the interrupt enable register. To disable the condition, reset the bit to 0 and write the byte mask to the interrupt enable register.

Note that Async Professional enables all four interrupt conditions. This is how it provides buffered input/output, automatic hardware flow control, and automatic detection of line errors and breaks.

Register 1 also has a second name (divisor latch high) and a second purpose. When the divisor latch access bit is set, register 1 becomes the high byte of the baud rate divisor (used for setting the baud rate). See the discussion of the divisor latch low register for more information on setting the baud rate.

Register 2: **interrupt identification register (read)**
 FIFO control register (write)

FIFO enabl	FIFO enabl	N/A	N/A	see below			int pend
7	6	5	4	3	2	1	0

Interrupt identification bits

This is the counterpart to the interrupt enable register. That is, once you've enabled the desired interrupt conditions (and received an interrupt) this register tells you which condition caused the

interrupt when more than one condition was enabled. Only the three least significant bits of the register are actually used, as shown below.

Since it is possible, even likely, that more than one condition may occur at the same time, bit 0 is used to determine whether all conditions that currently exist have been handled. When bit 0 has a value of 0, there are conditions waiting to be handled. When bit 0 has a value of 1, all outstanding conditions have been handled. Bits 3, 2, and 1 taken together identify the cause of the interrupt.

Because multiple conditions may occur at the same time, the UART presents the conditions in a prioritized order. The following table shows the priority used by the UART and the corresponding bit masks:

bits 3-0	priority	interrupt type
0 0 0 1	none	none
0 1 1 0	highest	line error or line break
0 1 0 0	second	received data available
1 1 0 0	second	received data available (FIFO timeout)
0 0 1 0	third	transmitter holding register empty
0 0 0 0	lowest	modem status change

When multiple interrupt conditions exist, the UART reports them one at a time in this prioritized order.

The FIFO timeout condition obviously occurs only on UARTs operating in FIFO mode. This interrupt condition is generated when received characters are waiting in the receive FIFO buffer below the trigger level and four character-times (the amount of time required to receive four characters at the current baud rate) elapse without receiving any new characters. For more details on FIFO operation, see the discussion of FIFO Buffering in §4.C.

In addition to providing information about pending interrupt conditions, this register also provides two FIFO status bits. These bits are *always* 0 for UARTs that don't possess FIFO buffers. They are both set for UARTs that possess the FIFO buffers, if currently in FIFO mode.

Register 2 doubles as a writeable register for enabling and disabling FIFO buffers. In its role as the FIFO control register, the 8 bits have the following meanings:

rcvr triggr high	rcvr triggr low	N/A	N/A	DMA mode	xmit reset	rcvr reset	FIFO enable
7	6	5	4	3	2	1	0

FIFO control bit definitions

The first step in specifying FIFO control information is always to set bit 0. This enables writing to the other FIFO control bits. When FIFO mode is enabled or disabled, the FIFO buffers are cleared of all data.

Writing a 1 to bit 1 clears just the receive FIFO buffer.

Writing a 1 to bit 2 clears just the transmit FIFO buffer. Async Professional doesn't use the transmit FIFO buffer.

Bit 3 is used to control DMA access. Async Professional doesn't use DMA access.

Bits 6 and 7 are used to specify the receive FIFO trigger level (the number of bytes stored in the FIFO before a receive interrupt is generated). The following table shows the possible trigger levels and the corresponding bit values:

bit7	bit6	trigger level
0	0	1
0	1	4
1	0	8
1	1	14

Register 3: line control register (write)

div latch access	send break	stick parity	parity type	enable parity	stop bits	data bits	
7	6	5	4	3	2	1	0

Line control bit definitions

The line control register is used to set the desired line parameters (baud rate, data bits, stop bits, and parity).

Bits 0 and 1 specify the number of data bits to use. The following table shows how these bits are interpreted:

bit1	bit0	data bits
0	0	5
0	1	6
1	0	7
1	1	8

Bit 2 specifies the number of stop bits to use. When bit 2 is set, two stop bits are generated and checked. When bit 2 is clear, one stop bit is generated and checked. (Note that when data bits is 5, setting bit 2 actually specifies 1.5 stop bits.)

Bit 3 specifies whether parity is used. If bit 3 is set, then a proper parity bit is transmitted after every data byte and checked for every incoming byte. If bit 3 is clear, then parity bits are not used at all.

Bit 4 specifies the parity type (odd or even), if any. If bit 4 is set, even parity is transmitted and checked. If bit 4 is clear, odd parity is used.

When bit 5 is set, either mark or space parity is used, and bit 4 determines which: bit 4 is set for mark parity and cleared for space parity. "Stick parity" means that the parity bit is "stuck," either on (mark) or off (space). If bit 5 is clear then neither mark nor space parity is used.

Putting bits 3-5 together allows you to see how the various parity types are specified:

bit5	bit4	bit3	parity type
0	0	0	None
0	0	1	Odd
0	1	1	Even
1	0	1	Space
1	1	1	Mark

Bit 6 is used to generate a line break. When this bit is set, the UART places zeros on the output line (generating a break).

Bit 7 is the divisor latch access bit described earlier. When this bit is set, register 0 and register 1 become the divisor latch registers which are used to set the desired baud rate.

Register 4: modem control register (write)

N/A	N/A	N/A	enable loopbk	OUT2 (reqd)	OUT1 N/A	enable RTS	enable DTR
7	6	5	4	3	2	1	0

Modem control register bit definitions

The primary purpose of the modem control register is to manage the DTR (Data Terminal Ready) and RTS (Request To Send) signals of the serial port. These two signals are also called the "handshaking" or "hardware flow control" signals. See "Hardware Flow Control" in §5.C for more information on the exact role of these signals.

Bit 4 enables an internal loopback that can be used to test some facets of proper UART operation.

Bit 3, OUT2, is a general purpose output signal that must always be enabled before interrupts will occur.

Bit 2, OUT1, is another general purpose output signal; however, it's not used in the PC architecture.

Bit 1 controls the state of the RTS signal. Writing a 1 into bit 1 raises the RTS signal. Writing a 0 into bit 1 lowers the RTS signal.

Bit 0 controls the state of the DTR signal. Writing a 1 into bit 0 raises the DTR signal. Writing a 0 into bit 0 lowers the DTR signal.

Register 5: line status register (read)

FIFO error	shift reg empty	hold reg empty	break recd	frame error	parity error	data ovrRun	char recd
7	6	5	4	3	2	1	0

Line status bit definitions

This register provides information about line error and line break conditions. It also provides the status of receive and transmit operations (i.e., whether a character was received and whether the UART is ready to transmit a character).

Generally, only line error and line break information is of concern to application programs. This information is read by Async Professional whenever a line error or line break is detected and an interrupt is generated.

Bit 7, the FIFO error status bit, is set if a character in the FIFO has a line error. Generally, you don't need to worry about this because the error is revealed in the normal fashion when the character is extracted from the FIFO buffer.

Bit 6, when set, indicates that the transmitter shift register is empty. The shift register, used internally by the UART, holds the character currently being transmitted while the individual bits are being shifted onto the output data line.

Bit 5, when set, indicates that the transmitter holding register, register 0, is empty. You should never write a character to register 0 unless this bit is clear. After a character is placed in the holding register, the UART moves it into the shift register (as soon as the character already in the shift register has been transmitted) and clears bit 5.

Bit 4 is set whenever a line break is received. This also causes a line error/break interrupt (which is how Async Professional knows that it received a break).

Bits 3 through 1, when set, indicate a line error has occurred. The nature of these line errors is discussed earlier in this chapter. All of these bits also generate interrupts (which is how Async Professional knows that the errors occurred).

Note that bits 4 through 1 are cleared when this register is read.

Bit 0, when set, indicates that characters are waiting in the receiver buffer register (register 0) and/or the receive FIFO. It remains set until all received data is read.

Register 6: modem status register (read)

DCD carr detect	RI ring indic	DSR data set rd	CTS clr to send	delta DCD	delta RI	delta DSR	delta CTS
7	6	5	4	3	2	1	0

Modem control register bit definitions

Just as the serial port can control the modem control signals, it can also read and report on the status of similar signals that are controlled by the attached device.

The modem status register actually provides two types of information. The most significant 4 bits (see the diagram) show the current state of the 4 modem signals. The least significant 4 bits indicate which, if any, of the signals have changed state (from zero to one, or vice-versa), since the last time the register was read. Async Professional updates its internal tables with these values as they change, and reports the results when asked to do so. You can test the signals individually or in combination using functions in OOCOM/APCOM (with the CheckXxx functions).

Obviously, all these signals assume that a modem is connected to the serial port and the cable contains all of the proper connections. Some of these signals are also used by non-modem devices to provide hardware flow control or another device-specific control functions.

Bit 7, data carrier detect (DCD), means that the local modem has established a connection to a remote modem. This bit remains set for as long as this connection is valid.

Bit 6, ring indicator (RI), is set whenever the phone is ringing (i.e., a call is coming in and needs to be answered).

Bit 5, data set ready (DSR), is generally set whenever a modem is attached and turned on. This assumes that the modem is configured to provide the DSR signal.

Bit 4, clear to send (CTS), is generally set whenever an attached modem is ready to receive data. This assumes that the modem is configured to provide the CTS signal.

Bit 3, delta DCD, is set whenever the DCD signal changes (from on to off, or off to on).

For reasons of symmetry, bit 2 is called the delta RI bit, but its proper name is trailing edge ring indicator. It is set, and generates an interrupt, on the first ring from an incoming call. (This is the most reliable way of checking for an incoming call.)

Bit 1, delta DSR, is set whenever the DSR signal changes.

Bit 0, delta CTS, is set whenever the CTS signal changes.

Interrupts

Interrupts in the 80x86 family come in two varieties: software and hardware. Software interrupts are generally used as a calling mechanism for various operating system services (e.g., your program issues software interrupts to DOS when it wants to read a file). Hardware interrupts are true, real-time interrupts from hardware devices that need servicing.

The IBM PC provides a total of eight prioritized hardware interrupts. It makes two of them available to serial ports. IBM ATs and PS/2s provide sixteen interrupts but still allocate only two standard interrupts to serial ports. The interrupt lines have the designations IRQ_n (where n is the number of the interrupt line). A vector number is associated with each IRQ. The vector refers to an interrupt service routine (ISR) that is called when that interrupt line is activated.

All PCs

IRQ	vector	purpose
0	8	timer
1	9	keyboard
2	A	reserved
3	B	Com2/Com4 (on PS/2s Com2..Com8)
4	C	Com1/Com3
5	D	2nd printer
6	E	diskette
7	F	printer

ATs and PS/2s only

IRQ	vector	purpose
8	70	real-time alarm
9	71	map to IRQ2
A	72	not assigned
B	73	not assigned
C	74	not assigned
D	75	not assigned
E	76	not assigned
F	77	not assigned

So what does all this mean to serial communications? Let's look at a couple of methods for handling received serial data. One option would be to poll the serial port. This means that the program sits in a loop and continually reads the line status register to see if a byte has arrived. When the byte arrives, the program reads it and goes back to the loop. Obviously such a scheme has serious drawbacks if the program must be able to do anything else.

Now let's look at the same situation using interrupts. First, we write an interrupt service routine (ISR) that, when called, extracts a received character from the UART and stores it in a previously allocated receive buffer. Then we program the UART to generate an interrupt whenever it receives a character (as discussed in the section on UART registers, above). Now the program can go on about its normal business without worrying about the serial port at all. Whenever it needs to check for input, it simply checks the receive buffer and retrieves the data from there.

Clearly this is the desired alternative. It's so handy in fact, that it is used for just about all serial port activity: receiving data, transmitting data, handling line errors and breaks, and providing hardware flow control.

C. Async Professional Architecture

Async Professional uses a layered architecture with some rather unusual features. Before exploring this architecture in detail, let's look at some of the goals we had in mind:

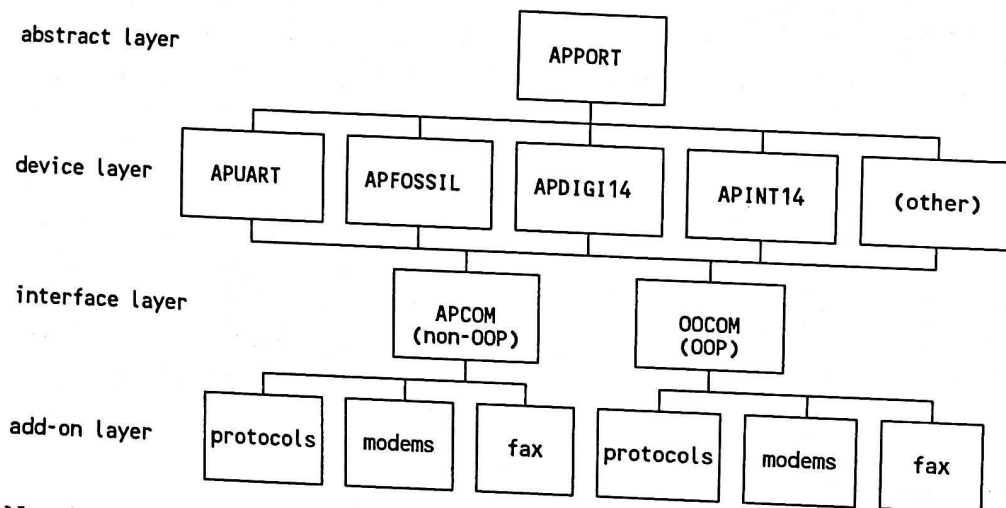
1. Offer device independence
2. Take advantage of object-oriented programming techniques
3. Provide a procedural interface

Device independence means that applications written using Async Professional should be easily extensible to environments using other serial port devices. Two examples of other serial port devices would be multiple port boards (like Digiboard) and network shared-modem pools.

We also wanted to take advantage of object-oriented programming techniques in Async Professional. We wanted to provide easy-to-use objects that would just plug into your program, and to provide the advantages of objects so you can extend them as needed.

This would have been easy were it not for our other desire to provide a non-object oriented interface to Async Professional. Experience has shown that many Turbo Pascal programmers prefer *not* to use objects and we wanted to make sure that they would still be able to use Async Professional.

Let's get a picture of our finished hierarchy in front of us so we can talk about how we provide both OOP and non-OOP interfaces.



Note the four layers of the architecture: abstract, device, interface, and add-on. These terms freely throughout the rest of this manual, so you should make sure that you understand them clearly now.

Abstract Layer

The abstract layer provides *only* the abstract data and procedure definitions. The most important data structure here, the PortRec, is a record that contains all of the fields that any type of port is likely to need (buffer addresses and sizes, line parameters, flow control options, and so on).

Another important function of the abstract layer is to define the "core routines." Core routines are the smallest set of building blocks necessary for an asynchronous communications library.

Let's go through the steps of building a small library of routines to see what this really means. Consider the process of receiving characters from the serial port. We would want our library to have a routine, which we'll call `GetChar`, that receives a single character. We might also want to have a procedure that returns an entire string; let's call this one `GetString`. And finally let's say we want something to scan the input stream for a particular character. We'll call it `WaitForChar`.

Do all of these routines need to access the serial port directly? Not at all. Upon inspection we see that, in fact, only `GetChar` actually needs to "go to the hardware." The other two routines, `GetString` and `WaitForChar` can be written using `GetChar`.

When writing `Async Professional`, we first picked the procedures and functions we wanted the library to contain. Working backwards from that, we came up with 29 routines that actually need to access the hardware. The other routines (over 300) can all be written using just those 29 core routines. We'll discuss the core routines in detail in `APPORT`, but it probably makes sense to take a first look at them here:

<code>InitPort</code>	- opens a com port with specified line options
<code>InitPortKeep</code>	- opens a com port with its existing line options
<code>DonePort</code>	- closes a com port
<code>SetUart</code>	- customizes UART addresses
<code>SetLine</code>	- sets/changes line parameters
<code>GetLine</code>	- returns current line parameters and status
<code>SetModem</code>	- sets/changes modem control lines
<code>GetModem</code>	- returns modem status
<code>GetChar</code>	- returns one received character
<code>PeekChar</code>	- peeks at one received character
<code>PutChar</code>	- transmits one character
<code>StartTransmitter</code>	- starts transmitting a stream of characters (internal use)
<code>CharReady</code>	- returns True if a received character is available
<code>TransReady</code>	- returns True if it's OK to transmit a character
<code>SendBreak</code>	- sends a line break
<code>ActivatePort</code>	- enables a device layer
<code>DeactivatePort</code>	- disables a device layer
<code>SavePort</code>	- saves the state of this com port
<code>RestorePort</code>	- restores the state of this com port
<code>BufferFlush</code>	- flushes the input and/or output buffers
<code>BufferStatus</code>	- returns buffer usage information
<code>HWFlowGet</code>	- returns the state of hardware flow control
<code>HWFlowSet</code>	- starts or stops hardware flow control
<code>SWFlowCtl</code>	- sets the on/off characters or resumes a blocked transmitter
<code>SWFlowGet</code>	- returns the state of software flow control
<code>SWFlowSet</code>	- starts or stops software flow control
<code>UpdateLineStatus</code>	- returns the line status register value
<code>UpdateModemStatus</code>	- returns the modem status register value
<code>GotError</code>	- reports an error to the user defined error handler

These core routines, and their parameter lists, are defined at the abstract layer. The abstract layer also defines a global procedure pointer for each core routine. These procedure pointers hold the addresses of the core routines, which are implemented in the device layer. The abstract layer is provided by one unit: `APPORT`. Every `Async Professional` program you write will need to include `APPORT` in its uses list.

Device Layer

The device layer is where the software meets the hardware. This is the *only* place, in all of `Async Professional`, where the library accesses the hardware directly.

The device layer implements the core routines that the abstract layer defines. Consider the example we introduced when talking about the abstract layer. We decided that we needed a core routine called GetChar and that its purpose would be to return one received character. We defined such a procedure type at the abstract layer but we didn't actually implement it there. Now our device layer must implement that procedure. This procedure will read the serial hardware and return the received character to the caller. The device layer must do this for all 29 core routines introduced at the abstract layer. The device layer must also initialize the global procedure pointers associated with the core routines.

Async Professional provides four device layers: APUART, APFOSSIL, APDIGI14, and APINT14.

APUART is the device layer unit you'll use most often (since it's the most capable). It provides interrupt-driven serial communications up to 115200 baud. It provides the complete Async Professional suite of features including automatic software flow control, automatic hardware flow control, and fully buffered input/output. To some extent, it also supports non-standard UARTs by letting you change the UART base address and IRQ line.

APFOSSIL allows you to use Async Professional with a FOSSIL driver. FOSSIL device drivers are popular among BBS programs. You may want to use APFOSSIL in applications that will run with a BBS that uses a FOSSIL driver. APFOSSIL provides a subset of APUART features: baud rates to 19200, some software/hardware flow control options, and buffered input/output.

APDIGI14 provides support for DigiBoard intelligent multiport boards that use the DigiBoard Universal Device Driver. It provides nearly the same set of features as APUART: baud rates to 115200, almost all software/hardware flow control options, and buffered input/output.

APINT14 is far less capable than the other device layers because it uses the serial services built into your PC's BIOS. You may find it handy for some very simple applications or in cases where you have a PC with a truly off-the-wall UART that APUART can't use.

By taking advantage of the architecture of Async Professional, you can define your own device layer units for non-standard UARTs, multi-port boards, or network modem-sharing systems.

Interface Layer

The interface layer contains the 70 or so procedures and functions that your application program typically uses when sending and receiving data. Going back to our example again, this is where you would find the routines GetString and WaitForChar. These routines would then call the GetChar core routine to actually get characters from the serial port.

The same concept applies to all of the routines in the interface layer. They call whatever core routines are necessary to accomplish their purpose. The interface layer routines *never* access the serial port directly.

We've said that the interface routines call the core routines. But how do they call them? We certainly wouldn't want the interface routines to call a routine in a specific device layer. That destroys our ability to provide device independence.

Instead, the interface routines call the global procedure pointers associated with the core routines. (Remember, these global procedures pointers were defined back at the abstract layer and initialized by the device layer.) This way, the interface layer uses whatever device layer last set the global procedure pointers.

The interaction between the device and interface layer is somewhat different when objects are used. Instead of relying on global procedure pointers, which are initialized by a device layer unit and

point to the appropriate core routines, a port object defines a corresponding set of virtual methods that simply call the correct device layer routine directly.

The interface layer units (APCOM and OOCOM) are discussed fully in chapter 5. For now, it's enough to understand where they fit in the "big picture."

Add-on Layer

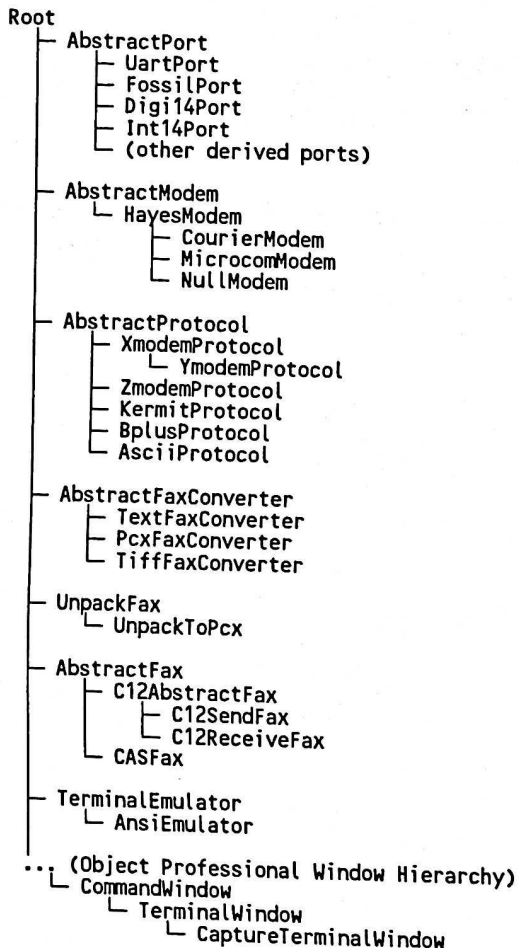
The add-on layer encompasses all Async Professional facilities that build upon the interface layer. This includes protocols, modem support, text file device drivers, and fax support.

Since these add-on units rely only on routines from the interface layer, they maintain the same device-independent property of the interface layer.

As with the interface layer units, the add-on layer units are fully discussed in their own chapters (Chapters 6, 7, 8, 9, 10). They are mentioned here so that you'll understand where they fit in the architecture.

D. Object Hierarchy

Specific objects in the hierarchy of Async Professional are described in detail in subsequent sections of this manual. What follows here is an overview of the hierarchy and a brief description of each object.



All objects are derived from Root. Root serves the same purpose here that it does in Object Professional—it assures that the virtual method table (VMT) link resides in the first two bytes of each object instance. This is essential for proper functioning of Object Professional's stream manager.

From Root, the hierarchy branches into five major object groups: ports, modems, protocols, emulators, and terminal windows. The last group, terminal windows, is actually derived from the Object Professional window hierarchy (which also has Root as its ultimate parent). These objects provide the OOP interface and add-on layers described in the previous section.

AbstractPort provides the data and routines needed to control a serial port including a variety of methods for opening and configuring a port, sending and receiving data, and for controlling various features (like hardware and software flow control).

UartPort, FossilPort, Digi14Port and Int14Port override low-level methods in AbstractPort to implement the physical aspects of how the derived port moves data between the serial port and the application. You would take the same approach when deriving your own port objects.

AbstractModem provides the data and routines needed to control a modem. This object is almost completely table driven and can be customized to nearly any degree.

The CourierModem, MicrocomModem, and HayesModem objects provide tables for controlling the U.S. Robotics 9600 BPS modems, some Microcom 9600 BPS modems, and all Hayes modems. The NullModem object provides for convenient modem testing when your PC is directly connected to another PC (i.e., you're using a null modem cable rather than a modem).

The AbstractProtocol provides the "boilerplate" functions used in file protocol transfers including file I/O, file name management, status tracking, and file logging.

The derived protocol objects, XmodemProtocol through AsciiProtocol, provide the block transfer methods specific to those protocols. The YmodemProtocol is similar enough to XmodemProtocol that they share many methods. In all other cases, however, the block transfer methods are unique to the derived protocol objects.

The AbstractFaxConverter object contains the basic fields and methods that all of the fax converters need. The TextFaxConverter reads ASCII files and converts each line to a series of raster lines before creating the fax image. The PcxFaxConverter and TiffFaxConverter unpack PCX and TIFF monochrome graphics files and convert each graphics raster line into a fax image raster line.

The UnpackFax object works with converted or received APF files, unpacking and passing each raster line back to your application.

AbstractFax provides the low-level data and functions required of all fax devices. Class 1/2 faxmodems require a few fields and functions that aren't required of CAS faxmodems, hence the middle level abstract object C12AbstractFax. Since many applications require just send or just receive capabilities, separate objects are provided (C12SendFax and C12ReceiveFax).

CASFax provides an interface between your application and the CASMODEM TSR.

The TerminalEmulator object defines the manner in which emulation information is returned to the application (but it does no emulation itself). The AnsiEmulator implements the IBM ANSI standard, as exemplified by ANSI.SYS.

The TerminalWindow object is not directly derived from an Async Professional object. Instead, it's derived from the Object Professional window hierarchy, so you must have Object Professional to use TerminalWindow class. See the Object Professional manual for more information.

CommandWindow is part of the Object Professional hierarchy. It's shown here because it's the immediate parent of a TerminalWindow. A CommandWindow is a window that includes all of the properties of a typical Object Professional window: it's moveable, resizable, stackable, supports frames, and includes a CommandProcessor for interpreting keyboard and mouse commands.

The TerminalWindow object adds support for serial I/O to the CommandWindow. All characters typed at the keyboard are also sent to the associated serial port (where they are transmitted to the remote computer). All characters received at the serial port are displayed in the TerminalWindow. The TerminalWindow includes a virtual screen which serves as a scroll-back buffer and maintains the contents of the TerminalWindow while that window is obscured.

The CaptureTerminalWindow adds support for capturing received data directly to a file.

E. Using Async Professional in Protected Mode DOS

Borland Pascal 7 introduced protected mode program generation to the Pascal programming environment. The benefit of protected mode programming is vastly improved capacity (code and data) for your program. Async Professional 2.0 is fully compatible with protected mode program generation. No changes or setup steps are required for protected mode programming. Simply select "protected mode" as your compilation target using the IDE or command line compiler and compile as usual.

Async Professional accesses protected mode services using an undocumented unit named DPMI. This unit centralizes certain variables and provides low-level tools for the DPMI (Dos Protected Mode Interface) API such as selector allocation/deallocation. You are free to use this unit in your application, but don't change any of its public declarations.

When running in protected mode, you may notice that programs using Async Professional run slower than a similar real mode program. The slowdown is due to the additional overhead of handling interrupts in a protected mode program. A protected mode program doesn't spend all of its time in protected mode. Most DOS or BIOS calls require the DPMI server to switch the processor from protected to real mode, call the DOS or BIOS function, then switch back to protected mode.

Assuming that your program will spend most of its time in protected mode, Async Professional installs protected mode interrupt service routines to handle the serial communication interrupts. However, if your machine *happens* to be in real mode when a character arrives at the port, the DPMI server will switch to protected mode, call the interrupt service routine, then switch back to real mode to finish what it was doing. This vastly increased overhead can lower the top baud rate your application can support on a given machine. For example, a real mode version of your program might be able to support a baud rate of 38400 baud. Under protected mode, that same program running on the same machine might experience occasional UART overruns at 38400 baud. If your application spends a lot of time in real mode, it might experience a large number of UART overruns at 38400 baud and you may have to reduce the baud rate to 19200 baud.

There are a variety of things you can do in your application and/or hardware to reduce this problem. First, try to minimize the amount of time your program spends in real mode. Since real mode switches are most often caused by DOS and BIOS calls, focus on minimizing the number of such calls your program makes. Pay special attention to what your program does when idling or waiting for keystrokes. Updating an onscreen clock using a DOS time call is particularly nasty because it causes nearly constant switches between protected and real mode.

Obviously, you can't eliminate all DOS/BIOS calls. If you find a lengthy one that you can't avoid, you can try imposing flow control to stop the remote from sending data during that operation. Hardware flow control is faster and more reliable than software flow control for this purpose. Call `SetRTS(False)` to lower your RTS signal, make the DOS call, then call `SetRTS(True)` to raise RTS again. The modem, or whatever is sending you data, will not send any data while it sees RTS low, thus preventing the possibility of your machine not being able to switch back to protected mode fast enough to handle a received character. (Before lowering RTS, check its current state; if it's low, make sure that you do *not* raise it after the operation.)

Second, try to identify system software that might also be spending a lot of time in real mode. Some known culprits are disk caches and disk doubling (compression) software. You may not be able to do anything to the system software, but you might be able to make some changes to your application to deal with it (like not writing to disk when your data is arriving or imposing flow control at your end).

Third, avoid installing protected mode timer or keyboard interrupt service routines. Real mode interrupt service routines for these two interrupts are always present. Adding your own protected mode service on top of that can result in multiple mode switches for each hardware interrupt.

Fourth, use the RotateIrqPriority procedure to give your communications interrupts a higher priority than timer and keyboard interrupts. This changes the interrupt hardware, so be sure to rotate the priorities back to normal before your program ends.

Fifth, set the FIFO trigger to a fairly low level, like 4 or even 1. This gives the UART a much higher tolerance for long delays. If you're not already using a UART with a FIFO buffer, then you may want to consider getting a 16550 UART chip.

If all else fails, you may simply have to select a lower baud rate. Usually stepping down one increment (from 38400 baud to 19200 baud, for example) is sufficient.

Note also that there are several different DPMI servers that your program may end up using (DPMI16BI from Borland, Windows when running in a Windows DOS box, QDPMI, etc.). These performance issues may vary from server to server since they will likely have different overhead times in servicing interrupts and performing mode switches.

All of the supplied example and demonstration programs are compatible with protected mode.

F. Protected Mode DLLs

Borland's protected mode services include Dynamic Link Libraries (DLLs)—libraries of code that are not linked directly into your program, but called by your program at runtime. One limitation of DLLs in Borland Pascal 7.0 is that they cannot export objects. Because of this limitation, only the procedural portions of Async Professional can be compiled to protected mode DLLs. (The OOP portions can still be compiled in protected mode, but they must be linked normally into your program.)

The Async Professional routines are split into three DLLs:

- APCOMD.DLL - device and interface layers
- APPROTD.DLL - all file transfer protocols
- APCOMP.DLL - ZIP/LZH compression/decompression routines

Two Async Professional units are not included in any DLL due to the infrequent use of these units: APMODEM and APTFDD.

To create the DLLs, assure that the {\$DEFINE UsePmodeDLL} statement is activated in APDEFINE.INC, then run Borland's MAKE utility:

```
MAKE -fAPRO
```

Async Professional provides a DLL import unit for each DLL:

- APCOMD.PAS
- APPROTD.PAS
- APCOMP.D.PAS

(Note: a .PAD extension is used for the DLL source code files to distinguish them from the import source code.)

To use these DLLs from your program, simply include the import units in your uses list instead of the usual list of Async Professional units. Here's how a typical uses list might look for a program using the device, interface, and protocol layers of Async Professional:

```
program PCom;  
uses  
    ApComD,  
    ApProtD;  
begin  
    ...  
end.
```

APCOMD.DLL, since it includes the device layer, honors whatever device layer settings are requested in APDEFINE.INC when the DLL is compiled. That is, if UseUart and UseDigi14 are the requested device layers when APCOMD.DLL is compiled, then only the functions of APUART and APDIGI14 will be available in that DLL—APFOSSIL and APINT14 will not be available.

The initialization block of the import unit APCOMD automatically activates a device layer. The device layer activated depends on which device layers are defined in APDEFINE.INC when the DLL and import unit are compiled. If only one device layer is defined in APDEFINE.INC, it is automatically activated. If multiple device layers are defined in APDEFINE.INC, one is activated automatically based on the following precedence:

1. UART
2. FOSSIL
3. DIGI14
4. INT14

If you want to activate a different layer, call one of the following routines provided by the import unit APCOMD:

```
{IFDEF UseUart}
procedure ActivateApUartDLL;
{ENDIF}
{IFDEF UseFossil}
procedure ActivateApFossilDLL;
{ENDIF}
{IFDEF UseDigi14}
procedure ActivateApDigi14DLL;
{ENDIF}
{IFDEF UseInt14}
procedure ActivateApInt14DLL;
{ENDIF}
```

Unlike directly linked protected mode units, using DLLs does require one small change to your program. You can no longer reference the global variables AsyncStatus or ArchiveStatus since they are now private to the DLL. Instead, two functions are provided to return the values of AsyncStatus and ArchiveStatus (APMISC):

```
function GetAsyncStatus : Word;
  {-Return the value of AsyncStatus, does not reset AsyncStatus}
function GetArchiveStatus : Word;
  {-Return the value of ArchiveStatus, does not reset ArchiveStatus}
```

Use these functions wherever you previously used AsyncStatus or ArchiveStatus:

```
ComPort^.ProtocolTransmit;
if GetAsyncStatus <> ecOk then
  ReportError(GetAsyncStatus);
```

Neither function resets the associated status variable; you can call each function as often as necessary and always get value set by the last Async Professional operation.

Two functions for setting AsyncStatus and ArchiveStatus are also provided:

```
procedure SetAsyncStatus(Status : Word);
  {-Set a new AsyncStatus value}
procedure SetArchiveStatus(Status : Word);
  {-Set a new ArchiveStatus value}
```

There are no normal circumstances that would require you to use these two procedures. They are provided for completeness and for unusual circumstances only.

3. Timer Functions

A. Overview

APTIMER provides the timer functions required internally by Async Professional. Since you might find these routines handy for your programs as well, they are available and documented here.

APTIMER's basic time unit is the BIOS clock tick. In case you're not already familiar with it, one clock tick is approximately 55 milliseconds. Or, put another way, there are about 18.2 clock ticks per second. This means that 55 milliseconds is the smallest interval that you can time. If a timed event takes zero clock ticks, all you know is that the actual event took something less than 55 milliseconds. One clock tick indicates that the event took something less than 110 milliseconds, and so on.

Unless otherwise specified, all Async Professional routines that have timeout parameters—the Timeout parameter to GetCharTimeout, for example—require a value expressed in terms of clock ticks. APTIMER interfaces routines to convert between ticks and seconds in case you prefer to work with seconds.

Async Professional generally uses timer functions to put a time limit on an event (for example, how long to wait for the next incoming character). For that reason, the basic record it works with is called an EventTimer. When you initialize an EventTimer, you specify the duration of the event. In some instances, though, you may be more interested in elapsed time than in whether or not a particular EventTimer has expired. As you'll see, EventTimers work equally well in both cases.

APTIMER also contains two routines for dealing with millisecond delays: Delay and CalibrateDelay. These routines are used solely by APASCII and OOASCII since those units need to specify very short delays.

Let's take a look at a program using some timer functions:

```
program ExTimer; {EXTIMER.PAS}
uses
  Crt, ApTimer;
var
  ET : EventTimer;
begin
  NewTimer(ET, Secs2Tics(60));
  WriteLn('Press any key to abort');
  repeat
    Write('M'Elapsed ticks: ', ElapsedTime(ET):5,
          ' Remaining ticks: ', RemainingTime(ET):5);
  until KeyPressed or TimerExpired(ET);
  WriteLn;
  if not KeyPressed then
    WriteLn('G, 'Timer expired')
  else
    WriteLn('G, 'Aborted');
end.
```

EventTimers are initialized by calling NewTimer and passing it an EventTimer and the number of ticks until expiration. This example uses the Secs2Tics function to start an EventTimer to expire in 60 seconds (or 1092 clock ticks).

This program loops continuously, displaying the elapsed ticks and the remaining ticks until the timer expires or a key is pressed.

What this program doesn't show is that it is perfectly legal to use the ElapsedTime functions even after a timer has expired. However, a timer is good only for 24 hours. If you reference a timer after 24 hours, the results are modulo 24 hours.

Declarations

Constants

SecsPerDay = 86400;

Number of seconds in one day.

TicsPerDay = 1573040;

Number of clock ticks in one day (based on 18.20648 ticks/second).

Types

EventTimer =

record

StartTics : LongInt; {tick count when timer was initialized}}

ExpireTics : LongInt; {tick count when timer will expire}

end;

The standard event timer record structure used by all timing routines.

Variables

CountsPerMs : Word;

This variable contains the loop count used by Delay to generate a 1 ms delay.

Syntax

```
procedure CalibrateDelay;
```

Purpose

Delay calibration routine.

Description

Recalibrates the timing count (CountsPerMs) used by Delay to generate a one millisecond delay. APTIMER automatically calls this routine within its initialization block. Recalibration is useful on some systems with a high clock speed that slow down the clock when accessing a floppy drive, and in situations where a multi-tasking environment such as DesqView is involved.

Syntax

```
procedure Delay(Ms : Word);
```

Purpose

Delay for specified number of milliseconds.

Description

This routine delays for approximately the specified number of milliseconds.

Example

```
Delay(1000);  
Delay for one second.
```

Syntax

```
procedure DelayTics(Tics : LongInt);
```

Purpose

Delay for a specified number of ticks.

Description

Similar in purpose to Delay, but uses clock ticks instead of milliseconds. DelayTics does nothing if Tics <= 0. It delays for at most one day.

Example

```
DelayTics(18);  
Delay for approximately one second.
```

ElapsedTime / ElapsedTimeInMSecs

Syntax

```
function ElapsedTime(ET : EventTimer) : LongInt;
```

Purpose

Return the elapsed time, in ticks, for this timer.

Description

This function returns the amount of time that has elapsed, in ticks, since NewTimer was called to initialize the specified EventTimer.

Example

See the example for ElapsedTimeInSecs.

See Also

ElapsedTimeInSecs

NewTimer

Syntax

```
function ElapsedTimeInMSecs(ET : EventTimer) : LongInt;
```

Purpose

Return the elapsed time, in milliseconds, for this timer.

Description

This function returns the amount of time that has elapsed, in milliseconds, since NewTimer was called to initialize the specified EventTimer.

Example

See the example for ElapsedTimeInSecs.

See Also

ElapsedTimeInSecs

NewTimer

Syntax

```
function ElapsedTimeInSecs(ET : EventTimer) : LongInt;
```

Purpose

Return the elapsed time, in seconds, for this timer.

Description

This function returns the amount of time that has elapsed, in seconds, since NewTimer was called to initialize the specified EventTimer. Partial seconds are ignored.

Example

```
var ET : EventTimer;
begin
  WriteLn('You have 20 seconds to press a key:');
  NewTimerSecs(ET, 20);
  repeat
    Write('M, ElapsedTimeInSecs(ET));
  until KeyPressed or TimerExpired(ET);
  WriteLn;
  if not KeyPressed then WriteLn('G, 'Time is up.');
```

```
end.
```

Displays the elapsed time until a key is pressed or 20 seconds have passed.

See Also

ElapsedTime

ElapsedTimeInMSecs

Syntax

```
procedure NewTimer(var ET : EventTimer; Tics : LongInt);
```

Purpose

Return an EventTimer that will expire in the specified number of clock ticks.

Description

This routine initializes an EventTimer record, which is used to measure elapsed time. NewTimer does two things: 1) it stores the current time in the StartTics field of ET, and 2) it calculates what the time will be when Tics number of clock ticks expire and stores that value in ExpireTics. Tics must be less than or equal to TicsPerDay (= 1,573,040).

Example

```
var ET : EventTimer;
begin
  WriteLn('You have 20 seconds to press a key. Time remaining:');
  NewTimer(ET, Secs2Tics(20));
  repeat
    Write('M, Tics2Secs(RemainingTime(ET)):2);
  until KeyPressed or TimerExpired(ET);
  WriteLn;
  if not KeyPressed then WriteLn('G, 'Timer expired');
```

```
end.
```

Displays the remaining time until a key is pressed or 20 seconds have passed.

See Also

ElapsedTime
NewTimerSecs

RemainingTime
TimerExpired

NewTimerSecs / RemainingTime

Syntax

```
procedure NewTimerSecs(var ET : EventTimer; Secs : LongInt);
```

Purpose

Return an EventTimer that will expire in the specified number of seconds.

Description

This routine is identical to NewTimer except that the timeout period is expressed in terms of seconds rather than clock ticks.

Example

```
var
  ET : EventTimer;
begin
  WriteLn('You have 20 seconds to press a key. Time remaining:');
  NewTimerSecs(ET, 20);
  repeat
    Write('M, RemainingTimeInSecs(ET):2);
  until KeyPressed or TimerExpired(ET);
  WriteLn;
  if not KeyPressed then
    WriteLn('^G, 'Timer expired');
end.
```

This program continuously displays the remaining time until a key is pressed or 20 seconds have passed.

See Also

ElapsedTimeInSecs
NewTimer

RemainingTimeInSecs
TimerExpired

Syntax

```
function RemainingTime(ET : EventTimer) : LongInt;
```

Purpose

Return the remaining time, in ticks, for this timer.

Description

This function returns the amount of time remaining, in clock ticks, until the specified timer expires. If the timer is expired, RemainingTime returns zero.

Example

See the example for NewTimer.

See Also

RemainingTimeInMSecs
RemainingTimeInSecs

TimerExpired

RemainingTimeInMSecs / RemainingTimeInSecs

Syntax

```
function RemainingTimeInMSecs(ET : EventTimer) : LongInt;
```

Purpose

Return the remaining time, in milliseconds, for this timer.

Description

This function returns the amount of time remaining, in milliseconds, until the specified timer expires. If the timer is expired, RemainingTimeInMSecs returns zero.

Example

```
var
  ET : EventTimer;
begin
  WriteLn('You have 20000 milliseconds to press a key. Time remaining:');
  NewTimerSecs(ET, 20);
  repeat
    Write('M, RemainingTimeInMSecs(ET):5, ' ms');
  until KeyPressed or TimerExpired(ET);
  WriteLn;
  if not KeyPressed then
    WriteLn('G, 'Timer expired');
end.
```

This program continuously displays the remaining time until a key is pressed or 20 seconds have passed.

See Also

RemainingTime

TimerExpired

Syntax

```
function RemainingTimeInSecs(ET : EventTimer) : LongInt;
```

Purpose

Return the remaining time, in seconds, for this timer.

Description

This function returns the amount of time remaining, in seconds, until the specified timer expires. If the timer is expired, RemainingTimeInSecs returns zero.

Example

See the example for NewTimerSecs.

See Also

RemainingTime

TimerExpired

Secs2Tics / Tics2Secs / TimerExpired

Syntax

```
function Secs2Tics(Secs : LongInt) : LongInt;
```

Purpose

Convert seconds to clock ticks.

Description

This function converts seconds to clock ticks.

Example

See the example for NewTimer.

Syntax

```
function Tics2Secs(Tics : LongInt) : LongInt;
```

Purpose

Convert clock ticks to seconds.

Description

This function converts clock ticks to seconds.

Example

See the example for NewTimer.

Syntax

```
function TimerExpired(ET : EventTimer) : Boolean;
```

Purpose

Return True if the specified EventTimer expired.

Description

This function returns True if the specified timer, previously initialized by NewTimer or NewTimerSecs, has expired.

Example

See the example for NewTimer.

See Also

NewTimer

NewTimerSecs

4. The Abstract and Device Layers

A. Overview

As discussed in Chapter 2, the Async Professional architecture consists of an abstract layer, a device layer, an interface layer, and an add-on layer. Although you will generally be working with Async Professional at the interface and add-on layers, you need a basic understanding of how the layers work together. Since some declarations and procedures are documented only at the abstract and device layers, knowing the roles of these layers will, at a minimum, help you navigate through the reference sections of this manual. This chapter covers the low-level units that make up the abstract and device layers of Async Professional (APPORT, APUART, APFOSSIL, APDIGI14, APINT14 and APMISC). The Core Routines, though not actually a separate unit, are also documented here.

APPORT is the abstract layer of Async Professional. It provides the data and procedural abstractions that the rest of the library builds upon. Of primary importance is the definition of the PortRec—the record that contains all of the fields a typical serial port is likely to need. The second major purpose of APPORT is to define the procedure types of the 29 core routines—those low-level procedures and functions that every device layer needs to support. Finally, APPORT contains the primary Async Professional debugging tool: tracing.

APUART is one of the supplied device layer units. In fact, it could be called the standard device layer since it is the one that nearly everyone will use. This is the only unit where Async Professional actually accesses the serial port hardware. It's designed to work with the serial ports as supplied on standard IBM PCs, compatibles, and PS/2s.

APFOSSIL is probably the second most popular device layer in Async Professional. It provides support for the standard FOSSIL device drivers used by many BBS systems. If you want to use the FOSSIL driver in your BBS programs and tools, you can do so with the APFOSSIL device layer. Unfortunately, FOSSIL device drivers don't provide nearly as much functionality as the APUART device layer. If you will be using a FOSSIL device driver with Async Professional, be sure to read about the limitations of FOSSIL in §4.D.

APDIGI14 is a supplied device layer that supports DigiBoard intelligent multiport boards using the DigiBoard Universal Device Driver (XIDOS5.SYS). Like the Universal Device Driver itself, APDIGI14 provides nearly all the capabilities of APUART. APDIGI14 supports the following DigiBoards: COM/Xi, PC/Xi, PC/Xe.

APINT14 is the final supplied device layer unit. Actually, it's not a very practical unit since it uses only the BIOS interrupt \$14 services. It is provided for two reasons: 1) for poorly compatible PCs that don't work with APUART; and 2) as a start on the network modem-sharing protocol based on interrupt \$14 services.

The Core Routines reference section describes those basic routines that every device layer must provide. Sometimes you call these routines directly. More often, you call higher level functions that build upon the core routines.

APMISC is a grab bag of general purpose routines needed by the rest of the library (e.g., string and bit manipulations). If you are using Object Professional or Turbo Professional, be sure to turn on UseOPro or UseTPro in APDEFINE.INC. This allows Async Professional to use routines in those libraries instead of from APMISC (saving as much as 3000 bytes of code and data).

Finally, the Async Professional approach to error handling is discussed in §4.I.

B. The Abstract Layer: APPORT

APPORT is the most basic building block in Async Professional. Nearly every other unit in the library builds upon APPORT. Your applications will also need to directly reference some of the types and constants provided by APPORT. In fact, nearly every program you write using Async Professional will need to have APPORT in its uses list.

APPORT provides the abstract definition of a serial communications port as used by Async Professional—what is called the PortRec. If you are using the object-oriented interface of Async Professional, you won't need to be too concerned about the PortRec since it is encapsulated within the port object. In the non-OOP interface, however, the PortRecPtr is your link to an open serial port—an initialized PortRecPtr is returned to you when you open a port and you pass that pointer to all subsequent routines.

APPORT also defines the procedure types of the 29 core routines. It doesn't actually do anything with them, however. The device layer actually implements these core routines. They are defined here so that all device layers, present and future, will follow the same format.

APPORT defines 29 global procedure pointers that correspond to the core routines. These global procedure pointers are set by the device layer and used by the non-OOP interface layer. This is how the non-OOP interface layer is linked to a particular device layer. These procedure pointers are initialized to small error stubs in APPORT so that if you forget to include a device layer unit in your program it won't just crash and burn—instead you get the error code ecNoDevice.

Throughout your Async Professional programs, you'll occasionally need to make reference to port related constants and data types (ComNames, port options, and so on). All such constants and data types are provided by APPORT. This is the primary reason that all of your programs need to have APPORT in their uses lists.

APPORT also provides a few facilities common to all ports: tracing (the primary debugging tool), error/status handling, and user abort procedures. This section also discusses, very briefly, what is involved in writing your own device layers (which would allow your applications to run on hardware that is not yet provided for).

It is possible to write a program using only core procedures. However, you are more likely to use both core procedures and the interface layer procedures from APCOM/OOCOM. And, of course, programs written using only core procedures won't be object oriented. In any case, here is an example of a program that uses only core procedures:

```
program ExCore;    {EXCORE.PAS}
uses
  Crt, ApMisc, ApPort, ApUart;
var
  P : PortRecPtr;
  C : Char;

begin
  {Open a port}
  InitPort(P, Com3, 2400, NoParity, 8, 1, 500, 500, DefPortOptions);
  if AsyncStatus <> ecOk then begin
    WriteLn('Failed to open port: ', AsyncStatus);
    Halt;
  end;
```



```

Delay(100);
PutChar(P, 'A');
PutChar(P, 'T');
PutChar(P, 'Z');
PutChar(P, cCR);
Delay(2000);

while CharReady(P) do begin
    GetChar(P, C);
    Write(C);
end;

DonePort(P);
end.

```

Notice that the uses list includes APMISC, APPORT, and APUART. This is the minimum uses list that any Async Professional program will have. When working with only the core routines, simply placing a device layer unit in the uses list establishes that unit as the active device layer (assuming that AutoDeviceInit is defined in APDEFINE.INC, as it is by default). Since this program has APUART in its uses list, APUART is the active device layer for this program.

This example program calls four core routines: InitPort, PutChar, GetChar and DonePort. These are actually the names of the global procedure pointers declared in APPORT. Since APUART has set these pointers to point to its routines, a call to InitPort is actually a call to APUART's implementation of InitPort. The same is true for PutChar, GetChar, DonePort, and all the other core routines.

After the port is open, this program uses PutChar to send a modem reset command (assuming that a modem is attached to COM3), waits two seconds for a response, then retrieves that response using GetChar. Finally, the port is closed with the call to DonePort.

Remember, the point of this example is to illustrate how the global procedure pointers link a particular device layer into your application. This is not a typical application, since most applications would use the services of APCOM or OOCOM as well. However, if your needs are simple and you want to minimize code space, the above approach would work just fine.

Tracing

In a perfect world all programs would work flawlessly as they were typed in. Since things rarely work out this nicely, it is often necessary to break out the debugging tools and apply some hard won debugging knowledge to get programs to behave themselves. Unfortunately, communications programs introduce some new debugging issues, and existing tools and knowledge may no longer be adequate.

For example, suppose you're writing a data collection program that regularly receives data from an instrument and writes that data to a database. While testing, you notice that a very small percentage of the data in the database is wrong. There are two broad explanations for such a problem: 1) the instrument sent you bad data; or 2) your program somehow corrupted good data before writing it to disk. Given that this applies only to a small percentage of records, you'd probably have to add some debugging code to your program that creates an audit report of all received data. Later you'd compare this audit report to the data in the database. If the data matched, then you would know the instrument sent bad data; if the data didn't match, you would know that your program corrupted the data. Either way, you'd now know what debugging steps to take next.

Variations of this need for an audit trail can occur in almost any communications program. Rather than force you to add such debugging code to your applications, it is built into Async Professional. This facility is called "Tracing."

Simply put, Tracing gives you the ability to keep track of all characters transmitted and received by your program. The goal of Tracing is to provide an audit report with a list of all transmitted and received data.

Every time your application successfully retrieves a received character (i.e., GetChar returns a character), a trace entry is created. Every time your application successfully sends a character to the transmitter (i.e., PutChar returns successfully), another trace entry is created.

These entries are stored in a circular queue of a size that you specify. Since it is a circular queue, it always contains the latest N transmitted/received characters (where N is the maximum number of entries the queue can hold).

At any time during your program, you can dump this trace queue to a file. The dump is a text file that contains a simple report of your application's handling of received and transmitted data. Here's a sample report:

Transmit:

**[24]B0100000027fed4[13][138][17]

Receive:

rz[13]**[24]B008000000dd38[13][138][17]

Transmit:

**[24]B0100000027fed4[13][138][17]

Receive:

[17]*[24]C[4][1][0][0][0][184]6[30][139]a.txx[0]6048 4734111064 0 0 3
18144[0][24]kP[215]B6

This report happens to be the first few exchanges of a Zmodem protocol transfer. Printable characters are displayed as such; non-printable characters are displayed as decimal values in square brackets (e.g., [13] is a carriage return).

Notice that the data is grouped in blocks of received and transmitted characters. This represents the flow of data as your program processed it. A new block is created whenever your program switches from transmitting to receiving or vice versa. If your program transmitted a continuous stream of data without ever stopping to receive data, then your report would consist of a single transmit block. Conversely, if your program were in a character-by-character terminal loop then each block might consist of only one character (because your program was constantly switching between sending and receiving single characters).

Note that this is entirely independent of the way the data may have arrived or departed from the serial port itself. For example, let's say you transmit the string 'ABCD' to a remote device that immediately starts echoing these characters back to you. In a true chronological report you'd see the received echo characters mixed in with, but slightly behind, the transmitted characters.

That's not what Tracing was designed to do. It was designed to show the data as your program processed it. The major benefit to such ordering is that it lets you check your program logic against the data it's working with. This type of tool was very useful during the development of the file transfer protocols. If a particular test went awry, it was a simple matter to review the trace and find out what went wrong (of course, finding out *why* it went wrong was another matter).

The mechanics of using Tracing are quite simple. Typical use would look something like this:

```
{Open a port}
```

```
InitPort(P, Com2, 1200, NoParity, 8, 1, 5000, 5000, DefPortOptions);
```

```

{Start tracing}
{$IFDEF Tracing}
InitTracing(500);
{$ENDIF}

{...work with the port...}

{Create the report}
{$IFDEF Tracing}
DumpTrace('TEST.TRC');
{$ENDIF}

{Close the port}
DonePort(P);

```

After opening the port, InitTracing is called to create a trace buffer of 500 entries. From then on, every successful call to PutChar and GetChar (really, every GetXxx and PutXxx routine) is automatically recorded. If more than 500 entries are generated, the trace queue always contains the last 500. The call to DumpTrace writes the report file to disk, turns off Tracing, and deallocates the trace buffer. We've adopted the convention of using the .TRC extension to identify trace files; however, you can use whatever extensions you like.

Notice that the calls to turn Tracing on and off are surrounded by conditional compilation directives. Since Tracing is a debugging tool, finished programs shouldn't carry around the extra debugging code. By default, Tracing is undefined in APDEFINE.INC. You'll need to specifically turn it on when you want to run traces. You can also use it, as above, to conditionally include the Tracing calls in your program. That way, turning it on in APDEFINE also turns it on in your program.

If, while reading this, you found yourself thinking that a true chronological report might be useful as well . . . we agree. The APUART unit has just such a facility, called EventLogging, and it's discussed in §4.C.

User Error Procedure

The principles behind error handling in Async Professional are described in §4.I. Basically, the process involves checking the global variable AsyncStatus after every procedure or function call. Since this can be a bit tedious, APPORT provides an alternative called the user error procedure.

This is a hook that all Async Professional routines call when they encounter an error (through an internal routine called GotError). The default error procedure does nothing. We strongly recommend, particularly during program development, that you install your own error handler. This way, you can see all errors, even if you neglected to check AsyncStatus after each Async Professional call.

It's easy to write and install a user error procedure:

```

{$F+}
procedure MyErrorProc(P : Pointer; var StatusCode : Word);
const
  Error1 = 'Error during async processing: ';
  Error2 = ' Press any key!';
  Blank = ' ';
var
  PR : PortRecPtr absolute P;
  C : Char;
  S : String;

```

```

begin
  with PR^ do begin
    {protocols handle transmission errors themselves}
    if ProtocolInProgress then
      Exit;

    if StatusCode mod 10000 <> 0 then begin
      Str(StatusCode, S);
      FastWrite(Error1+S+Error2, 25, 1, $07);
      C := ReadKey;
      FastWrite(Blank, 25, 1, $07);
    end;
  end;
end;
...
SetErrorProc(P, MyErrorProc);

```

This example shows the non-OOP calling sequence. Notice that the first parameter of the error procedure is an untyped pointer that this example treats as a PortRecPtr. When using objects, the declaration of this procedure is the same but the meaning of the P pointer is different—it's a pointer to an AbstractPort object. Here's how the same error procedure would look using objects:

```

{$F+}
procedure MyErrorProc(P : Pointer; var StatusCode : Word);
var AP : AbstractPortPtr absolute P;
begin
  with AP^ do begin
    ...
  end;
end;
...
ComPort^.SetErrorProc(MyErrorProc);

```

A few words on Async Professional error codes are appropriate here. Codes are the sum of an error prefix, which is a multiple of 10000, and an error number, which ranges from 0 to 9999. An error number of 0 means "no error." An error prefix of 0 is a fatal error (e.g., an object's constructor failed), and 10000 signifies a non-fatal error (e.g., a character could not be transmitted). Async Professional may add other prefixes in the future.

Notice the check for ProtocolInProgress. Since all protocols have specific error handling rules built into them, you typically won't want to interfere with that. Hence, when a protocol is in progress, your error procedure should just exit without doing anything. The only reason Async Professional passes such errors to the error handler at all is to facilitate debugging when implementing a new protocol.

This error procedure builds an error message from the error code, displays it, and waits for a key press. You may want to get fancier in your error procedures by adding pop-up windows or perhaps ignoring some errors (for instance, you may not want to report line errors here but let your application handle them instead).

The procedure SetErrorProc is used to install the error handler. Each port can have a unique error handler, or all ports can share one error handler.

A valid error procedure must be declared FAR using {\$F+} or the far keyword, it must not be nested inside any other procedure, and it must take exactly the parameters shown in the examples above.

Even if you have a user error procedure in place, you still need to make specific checks of AsyncStatus after many Async Professional calls. Usually this is when you are looking for some completion information rather than a particular error (e.g., after a protocol transfer, dialing a modem, and so on).

Finally, note that the error/status handling facilities of Async Professional serve two purposes. One, they help you catch "programmer errors" that crop up as you write your program (such as using some Async Professional routine incorrectly). Two, they help your finished program handle the routine errors (line errors, disk full, and so on) that are part of any communications application.

User Abort Function

Most Async Professional routines perform an action and return immediately. Some, however, are designed to wait for a condition to be met or to perform a lengthy process. The user abort function is a port-level hook that you can use to abort such processes. Generally, there are only two types of processes you might need to abort: 1) an Async Professional routine that is waiting for a condition to be met, and 2) protocol transfers.

Examples of routines that you might need to abort are GetCharTimeout and WaitForString. In each case, the routine sits in a loop waiting for a condition to become true (or timeout). A well behaved program needs to have a way to abort these routines.

The same thing applies to all file transfer protocols. Once you initiate a protocol transfer via ProtocolTransmit or ProtocolReceive, those routines perform the work of the protocol. Circumstances might arise where you need to cancel that protocol (e.g., the remote computer has a problem or the user decides to abort). Rather than providing a special protocol hook, the protocol uses its port's abort hook.

The abort function is a Boolean function that should return True when the current process is to be aborted. Otherwise it should return False. For example:

```
{ $F+ }
function KeyboardAbort : Boolean;
  {-Aborts work-in-progress if <Escape> is pressed}
var C : Char;
begin
  KeyboardAbort := False;
  if KeyPressed then begin
    KeyboardAbort := (ReadKey = cEsc);
    if C = #0 then C := ReadKey;
  end;
end;
...
SetAbortFunc(P, KeyboardAbort);
```

The abort function takes no parameters, so it's exactly the same for OOP and non-OOP formats. The only difference is the way the function is installed. The above example shows the non-OOP format. When using objects, the call to install the abort function would look something like this:

```
ComPort^.SetAbortFunc(KeyboardAbort);
```

This particular abort function returns True when the <Esc> key is pressed, indicating that the current process should be aborted (whether it's a protocol or an Async Professional routine in a wait loop).

A keyboard action isn't the only abort cue you might use. If you are using a modem, you might also want to check for loss of carrier signal. That is, your abort function would return True if the modem

status signal DCD (data carrier detect) isn't present—meaning that the phone connection between the local and remote modems was broken. The current process, whether a protocol transfer or a routine in a wait loop, would eventually timeout and return an error. However, catching this loss-of-carrier condition in your abort function could greatly speed things up.

Each port can have its own abort function or all ports can share the same abort function. The abort function must be declared FAR using {F+} or the far keyword, it must not be nested inside any other procedure, and it must be declared as shown in the example above.

Writing Your Own Device Layer

It is likely that 99% of Async Professional users will use only the standard serial ports provided by PC and AT compatibles and PS/2s. These standard ports are supported by the device layer unit APUART.PAS. If that's all you need, then you can skip this section since it deals only with the issues of writing your own device layer units.

If you are interested in running your applications on other serial devices (like network shared-modem pools) then you must write your own device layer unit. While Async Professional was designed with this eventuality in mind, it is unreasonable to describe in detail all of the issues to be considered (mostly because it would take far more time and space than is justified for such a low-use feature).

The best advice is to make sure that you thoroughly understand the abstract/device/interface layers of the Async Professional architecture described in Chapter 2. Then study the source code to both APPORT and APUART. Get familiar with the core procedures defined by APPORT and then look at APUART's implementation for each of these.

The basic rules are quite simple. You must get all of your data to fit within the standard PortRec. Hopefully, you can use the existing fields in PortRec in the same fashion that APUART uses them. If not, then you must allocate a sub-record and put its address in the UserData field of the PortRec.

You must also define and implement a procedure within your device layer unit for each core procedure pointer defined by APPORT. The implementation of each of these procedures is specific to the particular hardware you are using. You also need an ActivateLayer routine to set the global procedure pointers to point to your routines.

If you are using the OOP interface, you would create a new object type derived from AbstractPort. The only methods you should need to override are those that correspond to the core routines.

Declarations

Constants

BadPortOptions : Word = ptHiIrq;

Port options intended strictly for internal use by Async Professional. ptHiIrq is set internally if you use an IRQ > 7.

```
cNul = #0;  (^@)
cSoh = #1;  (^A)
cStx = #2;  (^B)
cEtx = #3;  (^C)
cEot = #4;  (^D)
cEnq = #5;  (^E)
cAck = #6;  (^F)
cBel = #7;  (^G)
cBS  = #8;  (^H)
cTab = #9;  (^I)
cLF  = #10; (^J)
```

```

cVT = #11; {^K}
cFF = #12; {^L}
cCR = #13; {^M}
cSO = #14; {^N}
cSI = #15; {^O}
cDle = #16; {^P}
cDC1 = #17; {^Q}  cXon = #17;
cDC2 = #18; {^R}
cDC3 = #19; {^S}  cXoff = #19;
cDC4 = #20; {^T}
cNak = #21; {^U}
cSyn = #22; {^V}
cEtb = #23; {^W}
cCan = #24; {^X}
cEM = #25; {^Y}
cSub = #26; {^Z}
cEsc = #27; {^[]}
cFS = #28; {^\\}
cGS = #29; {^]}
cRS = #30; {^_}
cUS = #31; {^_}

```

Convenient character constants (and aliases).

```

DataReadyMask      = $01; {Receive char is ready}
OverrunErrorMask   = $02; {Overrun error received}
ParityErrorMask     = $04; {Parity error received}
FramingErrorMask    = $08; {Framing error received}
BreakReceivedMask   = $10; {Break received}
THREMask           = $20; {Transmitter holding register is empty}
TEMask             = $40; {Transmitter is empty}
FIFOErrorMask       = $80; {FIFO error received}

```

These bit masks describe all of the bits in a UART line status register. Generally, you should not need to refer to these.

```

DefaultLineOptions : LineOptionRecord = (
    Parity : NoParity;
    DataBits : 8;
    StopBits : 1;
    Options : DefPortOptionsSimple;
    InSize : 2048;
    OutSize : 2048 + 30);

```

The InitPortFast procedures and Init constructors use these default values when opening ports. If your port can use these values exactly, then just call InitPortFast or Init in your application. If you need other values, you can either change this record constant or use the InitPort/InitCustom routines.

```

DefaultXonChar : Char = cXon; {Standard Xon char (DC1)}
DefaultXoffChar : Char = cXoff; {Standard Xoff char (DC3)}

```

Software flow control is also referred to as Xon/Xoff flow control—the Xon and Xoff characters start and stop the transmitter. However, there are some devices that may use other flow control characters, so Async Professional allows for that possibility. Since 99% of the time Xon and Xoff are suitable, these constants refer to them. If your application will always use other characters, you might find it more convenient to modify these typed constants rather than calling SWFlowSetChars for every serial port.

```

DefPortOptionsSimple = ptReturnPartialGets+ptReturnDelimiter+
    ptExecutePartialPuts+ptDropModemOnClose+
    ptRaiseModemOnOpen+ptRestoreOnClose;
DefPortOptions : Word = DefPortOptionsSimple;

```

Default port options.


```

DeltaCTSMask = $01; {CTS changed since last read}
DeltaDSRMask = $02; {DSR changed since last read}
DeltaRIMask = $04; {RI changed since last read}
DeltaDCDMask = $08; {DCD changed since last read}
CTSMask      = $10; {Clear to send}
DSRMask      = $20; {Data set ready}
RIMask       = $40; {Ring indicator}
DCDMask      = $80; {Data carrier detect}

```

These bit masks describe all of the bits in a UART modem status register. Generally, you should not need to refer to these.

```

DTRMask      = $01; {Data terminal ready}
RTSMask      = $02; {Request to send}
Out1Mask     = $04; {Output bit 1}
Out2Mask     = $08; {Output bit 2}
LoopbackMask = $10; {Loopback testing}

```

These bit masks describe all of the bits in a UART modem control register. Generally, you should not need to refer to these.

```

hfUsedDTR    = $01; {Use DTR for receive flow control}
hfUseRTS     = $02; {Use RTS for receive flow control}
hfRequireDSR = $04; {Require DSR before transmitting}
hfRequireCTS = $08; {Require CTS before transmitting}
hfDTRActiveLow = $10; {Make DTR active low}
hfRTSActiveLow = $20; {Make RTS active low}
hfDSRActiveLow = $40; {Make DSR active low}
hfCTSActiveLow = $80; {Make CTS active low}

```

Hardware flow control options. See HWFlowEnable (APCOM/OOCOM) in §5.C for details.

```
MaxActivePort = 36;
```

The maximum number of simultaneously active (open) ports that can be managed by APPORT. Note that the actual limit on the number of simultaneously open ports is set by the device layer (in the case of APUART, the limit is four).

```

MinInBuff = 10;
MinOutBuff = 10;

```

The minimum acceptable sizes for input and output buffers, respectively. Requests for smaller buffers will fail.

```

NoDevice      = 0;
UartDevice    = 1;
Int14Device   = 2;
FossilDevice  = 3;
Digi14Device  = 4;

```

Constants representing the currently supported device types.

```

ParityString : array[ParityType] of String[5] = (
  'None', 'Odd', 'Even', 'Mark', 'Space');

```

An array of strings describing all possible parity settings. This is useful if your application needs to display the current parity setting.

```
PS2DetectMode : PS2Mode = PS2Auto;
```

Controls how port open routines interpret requests to open ports Com3 and Com4. The default is to try to detect whether the current machine is a PS/2 or AT compatible. See the type declaration for PS2Mode later in this section for more information.

```

ptReturnPartialGets = $0001; {True to return partial strings}
ptReturnDelimiter   = $0002; {True to return delimiter char}
ptExecutePartialPuts = $0004; {True to send partial blocks}
ptIgnoreDelimCase   = $0008; {True to ignore case on DelimSets}
ptRestoreOnClose     = $0010; {True to restore UART on close}

```



```

ptDropModemOnClose   = $0020; {True to drop modem signals on close}
ptRaiseModemOnOpen   = $0040; {True to raise modem signals on open}

```

Port options that are used by all device layers. See the appropriate device layers for other, specialized port options.

If the `ptReturnPartialGets` option is on (the default), the various `GetBlock` routines return whatever data is in the input buffer if the requested amount of data is not available. If the `ptReturnDelimiter` option is on (the default), the `GetString` and `GetStringTimeout` routines return the delimiter character that was found, as well as the string it delimits. If the `ptExecutePartialPuts` option is on (the default), the various `PutBlock` routines store as much data as possible in the output buffer if there is insufficient space to store the specified data.

If the `ptIgnoreDelimCase` option is off (the default), then `StringReady` and other routines that work with sets of delimiter characters treat the characters in those sets in a case-insensitive fashion. If the `ptRestoreOnClose` option is on (the default), the UART is restored to its original state when a port is closed. If the `ptDropModemOnClose` option is on (the default), the modem signals are dropped when a port is closed. If the `ptRaiseModemOnOpen` option is on (the default), then the DTR and RTS signals are raised when a port is first opened (see the entry for the core procedure `InitPort` in §4.G).

Note that there is a slight conflict between `ptRestoreOnClose` and `ptDropModemOnClose`; the first wants to restore the modem signals to whatever they were before the port was opened, the second wants to drop the modem signals regardless of what they were when the port was opened. If both of the options are specified, `ptDropModemOnClose` controls the modem signals. That is, all of the port conditions are restored *except* the modem control signals—these are always dropped.

```

ReceiveIntMask = $01; {Interrupt on received data}
TransmitIntMask = $02; {Interrupt on THR empty}
LineIntMask    = $04; {Interrupt on line status change}
ModemIntMask   = $08; {Interrupt on modem status change}

```

These bit masks describe all of the bits in a UART interrupt enable register. Generally, you should not need to refer to these.

```

sfReceiveFlow = $01; {Use receiver flow control}
sfTransmitFlow = $02; {Use transmitter flow control}
DefSWFOpt     = sfReceiveFlow + sfTransmitFlow;

```

Software flow control options. See `SWFlowEnableOpt` (APCOM/OOCOM) in §5.C for more information.

```

THreg = 0; {Transmitter holding register}
RDreg = 0; {Read data register}
BRLreg = 0; {Baud rate, least significant byte}
BRHreg = 1; {Baud rate, most significant byte}
IEReg = 1; {Interrupt enable register}
IIDreg = 2; {Interrupt identification register}
LCreg = 3; {Line control register}
MCREg = 4; {Modem control register}
LSreg = 5; {Line status register}
MSreg = 6; {Modem status register}
Sreg = 7; {Scratch register}

```

These constants reflect the UART register designations. Generally, you should not need to refer to these.

```

{$IFDEF Tracing}
TracingOn : Boolean = False;

```

This variable is intended for internal use. It is set to `True` when tracing has been enabled by a call to `InitTracing` or `StartTracing`.

```

WordLenOMask = $01; {Word length select 0}
WordLen1Mask = $02; {Word length select 1}
StopBitsMask = $04; {Number of stop bits}

```

```

ParityEnableMask = $08; {Parity enable}
EvenParityMask   = $10; {Even parity select}
StickParityMask  = $20; {Stick parity select}
SetBreakMask     = $40; {Set break}
DLABMask         = $80; {Set divisor latch access}

```

These bit masks describe all of the bits in a UART line control register. Generally, you should not need to refer to these.

Types

```
AbortFunc = function : Boolean;
```

A user-defined abort function. See "User Abort Function" earlier in this section. Also see SetAbortFunc (APCOM/OOCOM) in §5.C.

```
ActivatePortProc = procedure(P : PortRecPtr; Restore : Boolean);
```

A core procedure that turns interrupts on or off for the specified port, optionally restoring the associated interrupt vector.

```
AsyncErrorProc = procedure(P : Pointer; var StatusCode : Word);
```

A user-defined error procedure. See the discussion of User Error Procedures earlier in this section. Also see SetErrorProc (APCOM/OOCOM) in §5.C.

```
BPtr = ^Byte;
```

Used for typecasting the I/O buffer arrays. You can use this if you ever need to reference the I/O buffers directly.

```
BufferFlushProc = procedure(P : PortRecPtr; FlushIn, FlushOut: Boolean);
```

A core procedure that flushes the input and/or output buffers.

```
BufferStatusProc = procedure(P : PortRecPtr;
                             var InFree, OutFree, InUsed, OutUsed : Word);
```

A core procedure that returns buffer usage information.

```
CharArray = array[0..MaxInt] of Char;
```

Used for typecasting blocks passed to the GetBlock/PutBlock functions. You can use this if you ever need to reference those blocks directly.

```
CharReadyFunc = function(P : PortRecPtr) : Boolean;
```

A core procedure that returns True if any received characters are available.

```
CharSet = Set of Char;
```

Used for defining sets of delimiter characters.

```
{IFDEF LargeComNameSet}
ComNameType = (Com1, Com2, Com3, Com4, Com5, Com6, Com7, Com8);
{ELSE}
```

```
ComNameType = (Com1, Com2, Com3, Com4, Com5, Com6, Com7, Com8,
               Com9, Com10, Com11, Com12, Com13, Com14, Com15, Com16,
               Com17, Com18, Com19, Com20, Com21, Com22, Com23, Com24,
               Com25, Com26, Com27, Com28, Com29, Com30, Com31, Com32,
               Com33, Com34, Com35, Com36);
```

```
{ENDIF}
```

Used to reference a serial port by its formal name. The LargeComNameSet option is provided for device layer modules that use more than eight serial ports. Com1 and Com2 refer to the industry standard definitions. Com5 through Com8 refer to the IBM standard for PS/2s. Com3 and Com4 refer to either the industry standard definition or IBM's standard. See the type declaration of PS2Mode for information on how Async Professional treats Com3 and Com4.

```
DataBitType = 5..8;
```

Type used for number of data bits. Acceptable values are 5 through 8.

```
DonePortProc = procedure(var P : PortRecPtr);
```

A core procedure responsible for closing the specified port and disposing of memory.

```
FlowCtlProc = procedure(P : PortRecPtr; OnChar, OffChar : Char;
    Resume : Boolean);
```

A core procedure that sets the on/off characters for software flow control or resumes a blocked transmitter.

```
FlowGetFunc = function(P : PortRecPtr) : FlowState;
```

A core function that returns the state of hardware or software flow control.

```
FlowSetProc = procedure(P : PortRecPtr; Enable : Boolean;
    BufferFull, BufferResume : Word; Options : Word);
```

A core procedure that starts or stops hardware or software flow control.

```
FlowState = (fsOff, fsClear, fsTransWait, fsRecWait, fsAllWait);
```

These are the possible states for hardware or software flow control. If flow control is off, the state is fsOff. If neither side of the transmission is waiting to send, the state is fsClear. If this side was asked to wait, the state is fsTransWait. If the remote was asked to wait, the state is fsRecWait. If both sides are waiting, the state is fsAllWait.

```
GetCharProc = procedure(P : PortRecPtr; var C : Char);
```

A core procedure that returns the next received character, or reports an error if none is available.

```
GetLineProc = procedure(P : PortRecPtr;
    var Baud : LongInt;
    var Parity : ParityType;
    var DataBits : DataBitType;
    var StopBits : StopBitType;
    FromHardware : Boolean);
```

A core procedure that returns the current settings for baud rate, parity, and data and stop bits for the indicated port. If FromHardware is False, the procedure simply returns the values stored in the PortRec.

```
GetModemProc = procedure(P : PortRecPtr; var DTR, RTS : Boolean);
```

A core procedure that obtains the current DTR and RTS settings directly from the hardware.

```
GotErrorProc = procedure(P : PortRecPtr; StatusCode : Word);
```

A core procedure that is called internally when an error occurs.

```
InitPortKeepProc = procedure(var P : PortRecPtr; ComName : ComNameType;
    InSize, OutSize : Word);
```

A core procedure responsible for opening a port. Unlike an InitPortProc, such a procedure does not change the current line settings or DTR/RTS.

```
InitPortProc = procedure(var P : PortRecPtr; ComName : ComNameType;
    Baud : LongInt; Parity : ParityType;
    DataBits : DataBitType; StopBit : StopBitType;
    InSize, OutSize : Word; Options : Word);
```

A core procedure responsible for opening a port according to the given specifications. It should initialize a pointer to a port record (or report an error).

```
LineOptionRecord = record
    Parity : ParityType;
    DataBits : DataBitType;
    StopBits : StopBitType;
    Options : Word;
    InSize : Word;
    OutSize : Word;
end;
```

Records of this type hold the default line options used by InitPortFast and Init (for port objects). See the entry for the typed constant DefaultLineOptions earlier in this section.

```
ParityType = (NoParity, OddParity, EvenParity, MarkParity, SpaceParity);
```

The acceptable values for line parity.

PeekCharProc = procedure(P : PortRecPtr; var C : Char; PeekAhead : Word);
 A core procedure that returns the received character in the PeekAhead position of the input buffer. It does not physically remove it from the buffer. This routine does nothing if the device layer isn't buffered.

PortRecPtr = ^PortRec;

PortRec = record

BaseAddr	: Word;	{Base IO addr of UART}
Flags	: Word;	{Option flags for port options}
InBuffLen	: Word;	{Length of input buffer}
InBuffCount	: Word;	{Current # of chars in buffer}
OutBuffLen	: Word;	{Length of output buffer}
OutBuffCount	: Word;	{Current # of chars in buffer}
LostCharCount	: Word;	{Number of lost characters}
SWFFull	: Word;	{Hi-water mark for xoff}
SWFResume	: Word;	{Lo-water mark for xon}
HWFFull	: Word;	{Hi-water mark for auto-handshaking off}
HWFResume	: Word;	{Lo-water mark for auto-handshaking on}
CurBaud	: LongInt;	{Baud rate}
InBuff	: BPtr;	{Addr of input buffer}
InHead	: BPtr;	{Addr of current head}
InTail	: BPtr;	{Addr of current tail}
InBuffEnd	: BPtr;	{Addr of end of buffer}
OutBuff	: BPtr;	{Addr of output buffer}
OutHead	: BPtr;	{Addr of current head}
OutTail	: BPtr;	{Addr of current tail}
OutBuffEnd	: BPtr;	{Addr of end of buffer}
StatBuff	: BPtr;	{Addr of status buffer}
StatHead	: BPtr;	{Addr of current status head}
StatTail	: BPtr;	{Addr of current status tail}
StatBuffEnd	: BPtr;	{Addr of end of status buffer}
PortName	: ComNameType;	{"Standard" name (COM1,COM2,...)}
Vector	: Byte;	{Vector number of UART interrupt}
IrqNumber	: Byte;	{Mask to turn our interrupt on}
IntMask	: Byte;	{Current UART interrupt enable}
CurrentPort	: Byte;	{Current active port number}
ISREntryPoint	: Byte;	{Entry point number into APUART.ASM}
ModemStatus	: Byte;	{Current modem status}
ModemControl	: Byte;	{Current modem control value}
LineStatus	: Byte;	{Current line status}
LineControl	: Byte;	{Current line control value}
SWFState	: Byte;	{Software flow control options}
SWFGotXoff	: Boolean;	{True if Xoff char received}
SWFSentXoff	: Boolean;	{True if Xoff char sent}
SWFOnChar	: Char;	{SW flow on character (def = \$11, Xon)}
SWFOffChar	: Char;	{SW flow off character (def = \$13, Xoff)}
BreakReceived	: Boolean;	{True if break received}
TxReady	: Boolean;	{True if transmitter is available}
TxInts	: Boolean;	{True if using transmit interrupts}
TxIntsActive	: Boolean;	{True if transmit ints are active}
Buffered	: Boolean;	{True if using buffer serial I/O}
UseStatusBuffer	: Boolean;	{True if using status buffer}
OldUart	: Boolean;	{True if UART is 8250 or 8250B}
CurParity	: ParityType;	{Parity}
CurDataBits	: DataBitType;	{Data bits}
CurStopBits	: StopBitType;	{Stop bits}
SaveChar	: Char;	{Last known char (used internally only)}
LastXmitError	: Byte;	{Reason for last failed xmit}
HWFTTransMask	: Byte;	{Mask to XOR modem status bits to zero}

```

HWFTransHonor   : Byte;           {Mask of required modem status bits}
HWFRecMask      : Byte;           {Mask of "on" modem status bits}
HWFRecHonor     : Byte;           {Mask of modem status bits we care about}
HWFRemoteOff    : Boolean;        {True if we have turned off the remote}
ISRActive       : Boolean;        {True if in debugging mode}
ProtocolActive  : Boolean;        {True if this port is doing a protocol}
DoneProc        : DonePortProc;   {DonePort proc for this port}
ErrorProc       : AsyncErrorProc; {Pointer to error procedure}
ErrorData       : Pointer;        {Pointer passed to error routine}
UserAbort       : AbortFunc;      {Hook for user (keyboard) abort}
OrigPortState   : PortSaveRec;    {Record for saving init port config}
end;

```

This is the complete record definition of a port. It is shown here for informational purposes only. You should not rely on this definition since future versions of Async Professional may alter it.

What you will use--quite frequently--is the type PortRecPtr. When you open a port, you pass in a variable of type PortRecPtr, say P. If the open is successful, P points to an open PortRec. Every procedure or function that acts on that port expects that PortRecPtr as the first parameter.

```

PortSaveRec = record
  PicMask : Byte;
  IER      : Byte;
  MCR      : Byte;
  LCR      : Byte;
  BRLR     : Byte;
  BRHR     : Byte;
  FIFO     : Byte;
  Trigger  : Byte;
  Vector   : Pointer;
end;

```

Used to store the data needed to save and restore the state of a com port.

```
PS2Mode = (PS2On, PS2Off, PS2Auto, PS2Ignore);
```

Defines how InitPort (and its counterparts) interpret requests to open ports Com3 or Com4. Two standards for these ports exist: the de facto standard followed by nearly all clone/compatible manufacturers, and the IBM standard introduced with the IBM PS/2. (See the constant declaration of PS2DetectMode earlier in this section.)

PS2On means that InitPort uses the IBM PS/2 definition for Com3 and Com4. PS2Off means that InitPort uses the de facto standard. PS2Auto (the default) means that InitPort will try to figure out if it's running on a PS/2. While this test is guaranteed never to fail for a PS/2, some compatibles may incorrectly report that they are PS/2s. To account for this, you may want to give your applications an option to force one mode or the other. PS2Ignore means that InitPort uses whatever custom port definition you specified with SetUart.

```
PutCharProc = procedure(P : PortRecPtr; C : Char);
```

A core procedure that sends one character to the transmitter.

```
SavePortProc = procedure(P : PortRecPtr; var PSR);
```

A core procedure that either saves or restores the state of a com port.

```
SendBreakProc = procedure(P : PortRecPtr);
```

A core procedure responsible for sending a break signal.

```
SetLineProc = procedure(P : PortRecPtr; Baud : LongInt; Parity : ParityType;
  DataBits : DataBitType; StopBits : StopBitType);
```

A core procedure responsible for setting the line options for an already opened port. It is called automatically when a port is opened to establish the initial settings.

```
SetModemProc = procedure(P : PortRecPtr; SetDTR, SetRTS : Boolean);
```

A core procedure responsible for setting the output handshaking signals.

```
SetUartProc = procedure(ComName : ComNameType; NewBase : Word;
                        NewIrq, NewVector : Byte);
```

A core procedure for a device layer that directly accesses a UART. It defines the base address, interrupt mask, and vector for a given com port name (Com1, Com2, etc).

```
StartTransmitterProc = procedure(P : PortRecPtr);
```

A core procedure that starts the transmitter. Generally, such a procedure is of use only to the device layer and the interface layer, and should not be called directly.

```
StopBitType = 1..2;
```

The acceptable values for number of stop bits. Acceptable values are 1 and 2. Note that if the number of data bits is 5, a stop bit value of 2 is interpreted as 1.5 stop bits.

```
TransReadyFunc = function(P : PortRecPtr) : Boolean;
```

A core procedure that returns True if it's OK for the application to transmit at least one character.

```
UpdateLineStatusFunc = function(P : PortRecPtr) : Byte;
```

A core function that returns the line status register value.

```
UpdateModemStatusFunc = function(P : PortRecPtr) : Byte;
```

A core function that returns the modem status register value.

Variables

```
ActiveComPort : array[1..MaxActivePort] of PortRecPtr;
```

This array is used internally by Async Professional to keep track of open com ports.

```
{IFDEF UseOOP}
InitPort          : InitPortProc;
InitPortKeep      : InitPortKeepProc;
DonePort          : DonePortProc;
SetLine           : SetLineProc;
GetLine           : GetLineProc;
SetModem          : SetModemProc;
GetModem          : GetModemProc;
GetChar           : GetCharProc;
PeekChar          : PeekCharProc;
PutChar           : PutCharProc;
StartTransmitter  : StartTransmitterProc;
CharReady         : CharReadyFunc;
TransReady        : TransReadyFunc;
SendBreak         : SendBreakProc;
ActivatePort      : ActivatePortProc;
DeactivatePort    : ActivatePortProc;
SavePort          : SavePortProc;
RestorePort       : SavePortProc;
GotError          : GotErrorProc;
UpdateLineStatus  : UpdateLineStatusFunc;
UpdateModemStatus : UpdateModemStatusFunc;
HWFlowSet         : FlowSetProc;
HWFlowGet         : FlowGetFunc;
SWFlowSet         : FlowSetProc;
SWFlowGet         : FlowGetFunc;
SWFlowCtl         : FlowCtlProc;
BufferStatus      : BufferStatusProc;
BufferFlush       : BufferFlushProc;
{ENDIF}
SetUart           : SetUartProc;
```

These are the global procedure pointers referenced by APCOM. They *must* be filled in by a device layer for APCOM to work. These global procedure pointers don't apply at all (except for SetUart) when using OOCOM if UseOOP is defined in APDEFIN.INC.

AbortTracing / AddTraceEntry

Syntax

```
{IFDEF Tracing}  
procedure AbortTracing;
```

Purpose

Stop tracing and destroy the trace buffer.

Description

This procedure aborts the tracing operation and deallocates the buffer created by InitTracing. Note that AbortTracing does *not* write the tracing data to disk before aborting the operation.

Example

See the example for InitTracing.

See Also

DumpTrace

InitTracing

Syntax

```
{IFDEF Tracing}  
procedure AddTraceEntry(CurEntry : Char; CurCh : Char);
```

Purpose

Add a trace entry to the global TraceQueue.

Description

This procedure can be called to add an entry to the trace buffer. The CurEntry parameter indicates the type of entry, normally either 'T' for Transmit or 'R' for Receive. CurCh is the character that was transmitted or received.

Although it is intended primarily for internal use by Async Professional routines that send and receive data, you can use AddTraceEntry to store additional data in the trace buffer. You might, for example, wish to add an entry in cases where tracing has temporarily been suspended by a call to StopTracing.

Example

See the example for InitTracing.

CheckForString

Syntax

```
function CheckForString(var Index : Byte; C : Char; S : String;  
                        IgnoreCase : Boolean) : Boolean;
```

Purpose

Check for a specified string in an incoming data stream.

Description

This routine checks for string S in an incoming data stream. Unlike WaitForString (APCOM/OOCOM), CheckForString does not wait for that string. Instead, you call CheckForString for each new character you receive and CheckForString returns immediately. CheckForString returns True if the last character passed to it completes the string; False otherwise.

Index is a byte variable that you *must* initialize to zero before the first call to CheckForString. After that, CheckForString increments or resets Index to keep track of the current search. Do not modify Index unless you are starting a search for a different string. Because Index is stored outside of CheckForString, you can have multiple simultaneous comparisons as the example below illustrates.

C is the next character in the incoming data stream that you want to compare. IgnoreCase, when True, tells CheckForString to ignore case while comparing.

CheckForString can be used in many places where WaitForString isn't appropriate. Perhaps the best example of this is scanning for the Zmodem autostart sequence in your terminal loop. You can't really use WaitForString in that situation because you need to continue to process or display the incoming data. Although it's possible to use WaitForString's WaitChar hook to accomplish this, CheckForString can provide a cleaner implementation.

Finally, although CheckForString is an "interface" like routine (meaning it would normally be found in APCOM or OOCOM), it is in APPORT. This particular routine doesn't have any port dependencies and therefore doesn't require different OOP and procedural versions.

Example

```
var  
  ComPort : UartPortPtr;  
  Index1, Index2 : Byte;  
begin  
  ...  
  Index1 := 0;  
  Index2 := 0;  
  repeat  
    if ComPort^.CharReady then begin  
      GetChar(C);  
      Write(C);  
      if CheckForString(Index1, C, 'rz'#[13], False) then  
        {...start a Zmodem download...}  
      if CheckForString(Index2, C, 'someotherstring', False) then  
        {...start some other triggered process...}  
    end;  
  
    if KeyPressed then begin  
      {...keyboard handling...}  
    end;  
  until Finished;
```

A typical terminal loop that checks for two different incoming strings: an autodownload sequence from Zmodem and some other string. Note that Index1 and Index2 are both set to zero before CheckForString is called the first time.

Syntax

```
{ $IFDEF Tracing }  
procedure ClearTracing;
```

Purpose

Clear the trace buffer.

Description

This procedure empties the trace buffer. Note that it does *not* write the data in the buffer to disk before emptying the buffer, as DumpTrace does.

Example

See the example for InitTracing.

See Also

DumpTrace

InitTracing

Syntax

```
function ComNameString(ComName : ComNameType) : String;
```

Purpose

Return a displayable name string for a serial port.

Description

Provides a convenient way to display the name of a serial port. Returns 'COM1' for the Com1 value, 'COM2' for Com2, and so on.

DumpTrace / DumpTraceHex

Syntax

```
{IFDEF Tracing}  
procedure DumpTrace(FName : PathStr);
```

Purpose

Write the contents of the trace buffer to a file.

Description

This procedure commits the data in the trace buffer to disk by storing it in the file specified by FName. After the file has been created, AbortTracing is called to terminate the tracing operation and dispose of the trace buffer.

Example

See the example for InitTracing.

See Also

AbortTracing

InitTracing

Syntax

```
{IFDEF Tracing}  
procedure DumpTraceHex(FName : PathStr);
```

Purpose

Write the contents of the trace buffer to a hex file.

Description

This procedure commits the data in the trace buffer to disk by storing it in the file specified by FName. The file is written in hex. For example, the receipt of 'HELLO <CR><LF>' would be written in the file as:

Receive:

[20] HELLO [20] [0D] [0A]

After the file has been created, AbortTracing is called to terminate the tracing operation and dispose of the trace buffer.

See Also

AbortTracing

InitTracing

Syntax

```
{$IFDEF Tracing}
procedure InitTracing(NumEntries : Word);
```

Purpose

Prepare a circular tracing queue.

Description

This procedure is called to start a tracing operation. When called, it tries to allocate a buffer large enough to hold NumEntries entries. If NumEntries is greater than 32760, it sets AsyncStatus to ecInvalidArgument. If there is not enough memory to allocate a buffer of the specified size, it sets AsyncStatus to ecOutOfMemory. Otherwise, it allocates the trace buffer and sets TracingOn to True. A tracing operation started with a call to InitTracing generally ends with a call to DumpTrace, as shown in the example program below.

Example

```
program ExTrace; {EXTRACE.PAS}
uses
  Crt, ApMisc, ApPort, ApUart, ApCom;
var
  P : PortRecPtr;
  ChIn, ChOut: Char;

  procedure Abort(Msg : String);
  {-Close port and halt}
  begin
    WriteLn(Msg, ': ', AsyncStatus);
    DumpTrace('TRACE.TRC');
    DonePort(P);
    Halt(1);
  end;

begin
  {Open a port}
  InitPort(P, Com1, 1200, NoParity, 8, 1, 500, 500, DefPortOptions);
  if AsyncStatus <> ecOk then
    Abort('Failed to open port');

  {Start event logging}
  InitTracing(500);
  if AsyncStatus <> ecOk then
    Abort('Failed to start event logging');

  WriteLn('Using Com1,N,8,1 at 1200 baud');
  WriteLn('Terminal ready. Press <Esc> or <^C> to Quit');
  WriteLn('Trace will be written to TRACE.TRC');

  {Simple terminal}
  ChOut := #0;
  repeat
    if KeyPressed then begin
      ChOut := ReadKey;
      if ChOut <> #0 then begin
        PutChar(P, ChOut);
        if AsyncStatus <> ecOk then
          Abort('Failed PutChar');
        if ChOut = ^M then
          WriteLn
        else if ChOut >= ' ' then
```

InitTracing

```
        Write(ChOut);
    end
    else case ReadKey of
        #13 :           {AltR = Resume tracing}
            StartTracing;
        #1E :           {AltA = Abort tracing}
            begin
                AbortTracing;
                Abort('Aborting');
            end;
        #1F :           {AltS = Stop tracing}
            begin
                AddTraceEntry('S', ' ');
                StopTracing;
            end;
        #2E :           {AltC = Clear trace buffer}
            ClearTracing;
    end;
end;

{Process chars received}
if CharReady(P) then begin
    GetChar(P, ChIn);
    if AsyncStatus <> ecOk then begin
        WriteLn('M`J'LineError');
        FlushInBuffer(P);
    end else
        Write(ChIn);
    end;
until (ChOut = #27) or (ChOut = ^C);
DumpTrace('TRACE.TRC');
DonePort(P);
end.
```

This program implements a simple terminal program that shows how to use the Async Professional tracing functions.

See Also

AbortTracing

DumpTrace

Syntax

```
function IsPS2 : Boolean;
```

Purpose

Return True if the current machine is a PS/2.

Description

This function returns True if the calling program is being run on an IBM PS/2. This information is needed in applications that must use com ports other than Com1 and Com2, since the PS/2s use non-standard definitions of the higher numbered com ports.

Note that while this test is guaranteed to work on PS/2s, it does not work with some XT/AT clones and compatibles. See the type declaration of PS2Mode for more information.

Example

```
function ValidPortNum(ComNum : Byte) : Boolean;  
const LastPort : array[Boolean] of Byte = (4, 8);  
begin  
    ValidPortNum := ComNum <= LastPort[IsPS2];  
end;
```

Function to validate a port number for AT compatibles and PS/2s (assuming that the last valid port for an AT is Com4 and for a PS/2 is Com8).

RotateIrqPriority

Syntax

procedure RotateIrqPriority(Irq : Byte);

Purpose

Rotate priorities to give Irq the highest priority at the Programmable Interrupt Controller (PIC).

Description

This routine alters the normal prioritization of hardware interrupts. The original XT's had only one PIC for IRQs 0-7. AT's and all later machines have two PICs: a master for IRQs 0-7 and a slave for IRQs 8-F. The slave PIC is attached to IRQ 2 of the master PIC. PC software then maps IRQ 9 to IRQ 2.

IRQ	vector	purpose		
0	8	timer		
1	9	keyboard		
2	A	slave PIC		
		IRQ	vector	purpose
		8	70	real-time clock
		9	71	map to IRQ2
		A	72	not assigned
		B	73	not assigned
		C	74	not assigned
		D	75	not assigned
		E	76	not assigned
		F	77	not assigned
3	B	Com2/Com4		
4	C	Com1/Com3		
5	D	2nd printer		
6	E	diskette		
7	F	printer		

The priority of interrupts (from highest to lowest) is as follows:

0, 1, 8, 9, A, B, C, D, E, F, 3, 4, 5, 6, 7

Since the timer and keyboard have higher priorities than the serial ports, their interrupt service routines can delay response to serial port interrupts. The keyboard interrupt (IRQ 1) is particularly troublesome since the BIOS can take several hundred microseconds to process a keystroke—leading to `ecOverrunErrors` at baud rates as low as 9600 on slower 286/386 machines.

`RotateIrqPriority` uses a feature of the PIC to provide a solution to this problem. It rotates the priority of the table to bring the IRQ you specify to the top. If `Irq` is between 0 and 7, the master PIC is rotated to bring the requested `Irq` to the top. If the `Irq` is between 8 and F, the slave PIC is rotated to bring the requested interrupt to the top of the slave PIC.

Assume that you are using Com2 at a high baud rate and want to make sure you have the highest priority. Since Com2 uses IRQ 3, you would call `RotateIrqPriority(3)`—giving IRQ 3 the highest priority. After that call, the IRQ priorities would be:

3, 4, 5, 6, 7, 0, 1, 8, 9, A, B, C, D, E, F

If you want to give an IRQ between 8 and F the absolute highest priority, you need to rotate *both* the master and slave PICs. For example, to make IRQ A the highest priority interrupt, call:

`RotateIrqPriority($A);` {Rotate slave PIC so A is at top}

`RotateIrqPriority(2);` {Rotate master PIC so 2 is at top}

The priority list after these two calls would be:

A, B, C, D, E, F, 8, 9, 3, 4, 5, 6, 7, 0, 1

Important! The IRQ priority assignments at the PIC are "write only." Async Professional doesn't keep track of your priority changes, nor can it determine that a change was made. You must restore the normal IRQ priorities before your program exits. Call `SetIrqPriority(0)` and `SetIrqPriority(8)` immediately after closing the port or in your program's exit procedure.

SetPS2DetectMode / StartTracing / StopTracing

Syntax

```
procedure SetPS2DetectMode(Mode : PS2Mode);
```

Purpose

Set PS2 detect mode.

Description

This procedure sets the PS2 detect mode. You need to use this only if you are using a protected mode DLL. See "Protected Mode DLLs" in §2.F for more information about running in a protected mode DLL. Also see PS2Mode in the Declarations earlier in this section.

Syntax

```
{$IFDEF Tracing}  
procedure StartTracing;
```

Purpose

Restart tracing after a call to StopTracing.

Description

This procedure resumes a tracing operation after it has been temporarily suspended by calling StopTracing.

Example

See the example for InitTracing.

Syntax

```
{$IFDEF Tracing}  
procedure StopTracing;
```

Purpose

Stop tracing temporarily.

Description

This procedure temporarily prevents data from being stored in the trace buffer. The tracing operation can be resumed by calling StartTracing.

Example

See the example for InitTracing.

C. UART Device Layer: APUART

APUART provides the connection between your application and the serial hardware itself. In this case, that hardware is assumed to be the industry standard serial ports found on nearly all PCs, ATs, compatibles, and PS/2s. On PC and AT compatibles, those standard ports include Com1 through Com4. On PS/2s, the standard ports include Com1 through Com8.

APUART accesses these standard ports via a buffered, interrupt-driven kernel and protects you from the dirty details of accessing serial ports. (Kernel refers to the serial port interrupt service routine and the Pascal code in the APUART core procedures.) As characters are received by the kernel, they are placed in an input buffer and remain there until you remove them (using GetChar or a related routine). When you transmit characters, APUART moves your data to its output buffer and then sends the data out as quickly as the serial port allows. Your high-level calls, such as PutChar, don't have to wait until the data is actually transmitted.

Since APUART is a device layer unit, its main routines are documented in §4.G (Core Procedures). However, this unit does offer some UART specific features—notably EventLogging and support for on-chip FIFO buffers. These features are discussed later in this section.

Note that the various options of APUART are controlled via conditionally defined symbols in the file APDEFINE.INC. See §1.C, "Configuration," for more information on these options.

EventLogging

The previous section described a built-in, general purpose debugging facility called Tracing. Tracing is available to all ports and port objects and provides a report of all data received and transmitted by your program. EventLogging is another built-in debugging facility; however, it is *much* different from Tracing. About the only similarity is that it also produces an audit report of serial data received and transmitted by your program. Beyond that the two facilities differ radically. First, EventLogging works at a much lower level—recording events at the serial port itself. Second, its data is time-stamped and presented in exact chronological order. And third, EventLogging is specific to the APUART unit (it's not provided by APFOSSIL, APDIGI14, or APINT14).

The goal of EventLogging is to produce a time-stamped, chronological audit trail of all events that occur in the APUART kernel. An "event" is one of the following:

- a character is received
- a character is sent to the serial port to be transmitted
- a character was just transmitted
- a line error or line break occurred
- a modem status change occurred

Each event consists of several "event records." At least one of these event records is a time stamp record. Other event records might be used to show various data (usually UART register values) for that event.

Such a report has an incredible amount of information, most of which you won't ordinarily care about. However, if you are having problems connecting to a device, or you are debugging some difficult timing problems, or you are working out the hardware handshaking requirements for an obscure device, event logging reports are probably just what you need. They can show you, microsecond by microsecond, exactly what is happening at the serial port. Note that in many cases you'll need the technical specifications of the UART to figure exactly out what's going on. (Some basic register information is provided in Chapter 2.)

The rest of the description of EventLogging gets rather thick with details. You might just want to skim it to familiarize yourself with its capabilities and only come back to it should you need to do low-level debugging.

Here's a sample of an EventLogging report file:

Event#	Time	ISR	Data	[Hex]	Event
1	2324.119	1			ISR entered
2		1		[CC]	interrupt ID
3		1	A	[41]	receive character
4	2324.196	1			ISR exited
...					
9	6163.903	1			transmitter call
10		1	A	[41]	transmit character
11	6172.100	1			ISR entered
12		1		[C2]	interrupt ID
13	6172.157	1			ISR exited
14	8687.604	1			ISR entered
15		1		[C6]	interrupt ID
16		1		[E5]	line status
17	8687.667	1			ISR exited
18	12983.388	1			ISR entered
19		1		[C0]	interrupt ID
20		1		[21]	modem status
21	12983.454	1			ISR exited

Each line of the report is a distinct event record and has its own event record number. The record numbers are assigned at report generation time and simply represent the physical order of the events in the queue. The time stamp is in milliseconds with the actual resolution being one microsecond. The ISR number is a sequential number assigned to each opened port. The first port opened is one, the second port opened is two, and so on.

The next two columns show the data, if any, associated with this event record. The data is shown in both ASCII and hexadecimal notation. When the value is between 0 and 32, and the event record is a transmitted or received character, an ASCII mnemonic is used (e.g., [CR] for carriage return). If the value is above 127, then the mnemonic [HH] is used for the ASCII representation. Sometimes the meaning of this data is obvious, such as when it's a received or transmitted character. Other times, you need UART register documentation to figure out what it is.

The final column is a short text description of the specific event record. This example report shows all possible event record types (as well as all possible events).

Notice that the event records are separated by blank lines into groups. Each group represents one "event." The following describes each specific event in detail.

The first event (event records 1-4) shows a received character. At 2324.119 milliseconds after event logging was started, the interrupt service routine (ISR) was called. This means that there was some activity at the UART that caused it to generate an interrupt. APUART's ISR was called to service that interrupt.

The next event record after entering the ISR *always* shows the interrupt identification value (from the UART IID register). Using the UART register bit definitions (from Chapter 2) you can figure out that the register value, [CC], is indeed a received character interrupt. Even easier, you could just look at the next event record. Usually the comment under the event heading tells you exactly what's going on. In this case it says "received character" and its data field shows exactly what was received—the character 'A'.

Finally, the last event shows that the ISR finished its processing at 2324.196 milliseconds after event logging started, 77 microseconds after the ISR was first entered.

The next two events (event records 9-13) show the transmission of a single character. This happens in two distinct stages. The first event shows the start of the transmit process (when the character is sent to the UART). The second event shows the completion of the transmit process (the character has left the UART and the UART is ready for the next character to transmit).

Event records 9 and 10 show the start of the transmit process. At 6163.903 milliseconds after event logging was started, a character was sent to the UART to be transmitted. The next event record shows that it was an 'A'. Note that this event type occurs only when a transmit request is made and there are no other characters in the output buffer. If there *were* other characters in the buffer, this particular event record wouldn't have occurred and the "transmit character" event record would have been part of some other event (most likely a transmit completion event).

Event records 11 through 13 show a typical transmit completion event, which occurs after every transmitted character. In this particular case there were no other characters in the output buffer to transmit. If there had been another character, this would be shown with a "transmit character" event record.

The next event (event records 14-17) shows a line status event. This occurs whenever there is a line status error (overrun, parity, or framing error) or a line break. As usual, the first and last event records show the ISR entry and exit times and the second event record shows the interrupt type. The third event record, number 16, shows the new value from the UART's line status register.

The final event (event records 18-21) shows a modem status event, which occurs when one of the modem status signals (DCD, RI, DSR or RTS) changes. Event record 19 shows the new value from the UART's modem status register.

EventLogging adds a large amount of overhead, in code and in execution time, to APUART's interrupt service routine (ISR). Therefore, you should only activate this feature while debugging your program. EventLogging is activated by selecting an ISR configuration that includes the EventLogging facility. See §1.C, "Configuration," for more information.

FIFO Buffering

The current generation of UARTs (chip designation NS16550A) has a very useful enhancement—small on-chip buffers that hold up to 16 received and 16 transmitted characters. Since they operate in a first-in-first-out fashion, they are commonly called FIFO buffers; and when these are enabled, the chip is said to be in FIFO mode.

When receiving characters, FIFO mode offers two major benefits. First, it greatly relaxes the urgency in responding to received character interrupts. When not in FIFO mode, a received character interrupt must be serviced within one character-time or the old character could be overwritten by the next incoming character. In FIFO mode, up to 16 character-times are allowed to respond to an interrupt since the FIFO can hold all 16 characters. This is useful in environments where interrupts might be left off for long periods of time (some task-switching systems, applications with critical code sections, and even some EMS drivers).

Second, FIFO mode reduces the number of interrupts required to handle received characters. When not in FIFO mode, every received character can potentially generate an interrupt. This adds up to a fair amount of overhead in just the entry and exit code for the ISR. In FIFO mode, the UART can be instructed not to issue an interrupt for every received character, but to wait until a preset trigger level is reached. Then the ISR retrieves all received characters from the FIFO at one time. This can greatly reduce the number of physical interrupts (and associated overhead) required to receive data.

Using FIFO mode with Async Professional is quite easy. All of the details are handled automatically; all you need to do is turn the mode on. Following is some example code that checks for the existence of FIFO buffers and turns FIFO mode on if they are found.

```
OOP:
  New(P, InitFast(Com1, 2048, 2048));
  if AsyncStatus <> ecOk then
    {handle error};
  BaseAddr := P^.GetBaseAddr;
  if ClassifyUart(BaseAddr, False) = U16550A then
    SetFifoBuffering(BaseAddr, True, 8);
  ...
  Dispose(P, Done);
  SetFifoBuffering(BaseAddr, False, 0);

non-OOP:
  InitPortFast(P, Com1, 2048, 2048);
  if AsyncStatus <> ecOk then
    {handle error};
  BaseAddr := GetBaseAddr(P);
  if ClassifyUart(BaseAddr, False) = U16550A then
    SetFifoBuffering(BaseAddr, True, 8);
  ...
  DonePort(P);
  SetFifoBuffering(BaseAddr, False, 0);
```

In each example, the port is opened using the Async Professional default port definition. Note that the Async Professional open routines never disturb the existing state of FIFO buffering. If FIFO buffering was already on before the port was opened, then it will still be on; if it was off then it will still be off.

The open routines save the state of the FIFO buffering and the close routines attempt to restore that state (if the `ptRestoreOnClose` option is on). We say "attempt" because the restoration may not be perfect. There is no way to read the trigger level of the FIFO buffer (possible values are 1, 4, 8, and 14). If the FIFO needs to be restored to the enable state, then Async Professional takes a middle-of-the-road approach and sets the trigger level to 8.

If you don't specify the `ptRestoreOnClose` option, be sure to manually disable FIFO mode before exiting your program. Not all communication programs work well (or at all) with the UART in FIFO mode. If you find that other communication programs aren't working right after you run an Async Professional application, the first thing you should suspect is that FIFO mode was left enabled.

After opening the port, the example calls the `ClassifyUart` function to determine what type of UART is installed in this machine. FIFO buffers are present *only* if the function returns `U16550A`. If such a UART is detected, `SetFifoBuffering` is called to enable FIFO mode.

The last parameter in `SetFifoBuffering` specifies the trigger level. This is the FIFO buffer level (the number of characters accumulated in the buffer) at which the UART generates an interrupt. Possible values are 1, 4, 8, and 14. Lower values result in more physical interrupts and more

overhead, but there is less chance of losing incoming characters because there is more space available in the FIFO for characters to be stored temporarily. Higher values result in fewer physical interrupts and less overhead, but more chance of losing incoming characters (the FIFO will overflow more quickly).

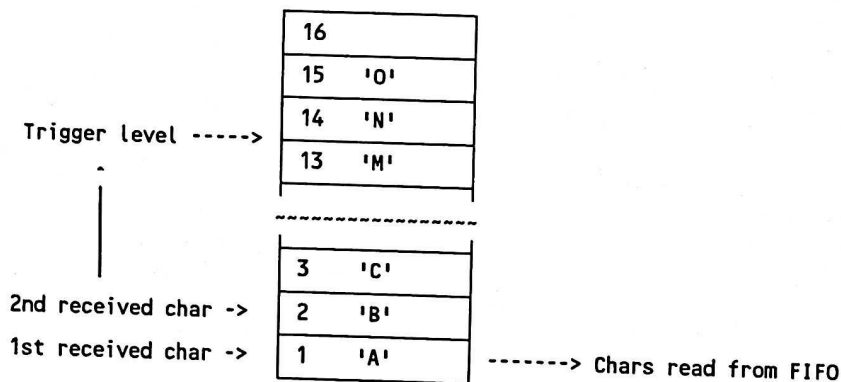
ClassifyUart and SetFifoBuffering do not take a PortRecPtr parameter nor are they methods of a port object. FIFOs are specific to UART devices; therefore, FIFO mode is not supported by the core routines. All FIFO support routines take instead a UART base address parameter (which can be retrieved by the GetBaseAddr function in APCOM/OOCOM).

After the call to SetFifoBuffering, the UART operates with fewer interrupts and is more tolerant of slow responses to its requests for interrupt servicing. Since a trigger level of 8 was specified, the UART will accumulate up to 8 characters before generating an interrupt. No other changes to the application are necessary.

If the ptRestoreOnClose option is set when the port is opened, as it is by default, then the Done (or DonePort) procedure attempts to restore the FIFO state. In this example, the FIFO mode is turned off *after* the port is closed. This ensures that the FIFO mode is now off even if it was on before the program started. This is the safest approach since many communication programs are not aware of FIFOs and will fail if the FIFOs happen to be left on.

That's about all you need to know to make use of the FIFO buffering capabilities of 16550A UARTs within Async Professional programs. This discussion continues with some of the inside details of FIFO buffering. Read on if you are curious.

To understand exactly how the FIFO works, let's look at a diagram of a buffer and step through a few received characters.



Assume that you turned on the FIFO and set the trigger level to 14. Let's say that you receive the 15 character string 'ABCDEFGHJKLMNO'. The 'A' is stored in the first slot of the FIFO, the 'B' is stored in the second slot, and so on. This continues until the 14th character ('N') is received. At that point, the FIFO has reached its trigger level and it generates a receive interrupt.

Let's further say that you are on a slow machine that's busy processing a higher priority network interrupt. So, before you get a chance to respond to the receive interrupt, another character comes in ('O'). Normally, this would be an overrun error, however, the FIFO just stores the character in the next slot. (Note that if this continued for two more characters, it *would* be an overrun error—there are limits even to the FIFO.)

Finally, the APUART interrupt service routine is given a chance to run. It sees that it has a receive character interrupt and it reads the first character ('A'). The rest of the characters all drop down one

slot. The FIFO is still filled above the trigger level so the ISR reads the next character ('B'). Again, the characters all drop down one slot. Now the FIFO is below the trigger level and the receive interrupt condition is cleared. A non-FIFO-aware ISR would quit at this point.

The APUART ISR, however, is aware that FIFO buffering is on and performs a further check to see if there are still characters in the FIFO. It will continue to extract characters until the FIFO is completely empty. What normally would have generated 15 separate interrupts is handled in 1 interrupt (eliminating all that extra overhead).

What if we don't get 14 bytes at once? Or ever? This is handled by the FIFO timeout interrupt. This interrupt is generated whenever there is at least one character in the receive FIFO and no new characters are received for four character-times. The timeout interrupt is treated just like a receive interrupt and the APUART ISR transfers all the characters in the FIFO to the port's input buffer.

The APUART ISR does not (at this time) make use of the transmit FIFO.

The FIFO buffer introduces one new wrinkle to handling line errors. Specifically, there is an extra step you may want to take when you encounter a line error. Normally, when you get a line error, you may decide to discard some or all of the characters currently in the input buffer. But what about the characters in the FIFO? Do they have errors, too? Should they be cleared just because there may have been a burst of noise on the line? The way you answer these questions depends on your application. However, you may want to check on the condition of the characters in the FIFO. The 16550 UART and Async Professional provide a way to do this.

The `CheckFifoError` function returns `True` if any of the characters in the FIFO has a line error. So, if you want to try to keep these characters, check this function. If you decide you don't want to keep these characters (on principle or because `CheckFifoError` returned `True`), then you should flush the FIFO. The easiest way to do this is to turn FIFO buffering off and back on again.

In practice, you might just want to skip this step. Because characters are only in the FIFO for a very short time, your error checking steps could just ignore the FIFO and handle the errors as you call `GetChar` (or `GetString`, `GetBlock`, etc.).

Declarations

Constants

```
DefBaseAddr : array[Com1..Com8] of Word = (  
    $03F8, $02F8, $03E8, $02E8, $4220, $4228, $5220, $5228);
```

The standard base I/O addresses for UARTs. Generally, you won't need to refer to these.

```
DefComVector : array[Com1..Com8] of Byte = (  
    $0C, $0B, $0C, $0B, $0B, $0B, $0B, $0B);
```

The standard interrupt vectors for UARTs. Generally, you won't need to refer to these.

```
DefIrqNumber : array[Com1..Com8] of Byte = (  
    4, 3, 4, 3, 3, 3, 3, 3);
```

The standard IRQ numbers for UARTs. Generally, you won't need to refer to these.

```
UartTypeString : array[UartType] of String[11] =  
    ('No UART', '8250B', '8250A/16450', '16550', '16550A');
```

A convenient set of strings for displaying UART types.

Types

```
UartType = (NoUart, U8250B, U8250A, U16550, U16550A);
```

The various types of UARTs recognized and classified by Async Professional.

ActivateApUart / ClassifyUart

Syntax

procedure ActivateApUart;

Purpose

Register APUART as the active device layer.

Description

You can have almost any number of device layer units in your program but, when using the non-OOP interface, only one may be active at any one time. This routine makes APUART the active device layer unit. Note that ActivateApUart is called automatically by APUART's initialization block if AutoDeviceInit is defined in APDEFINE.INC. If your program uses multiple device layer units, you should call ActivateApUart to insure that APUART is the active one. You would also call this if your application supports changing the device layer unit at run time.

Example

```
ActivateApUart; {make the APUART unit active}
{... use other Async Professional calls ...}
ActivateApInt14; {make the APINT14 unit active}
```

Shows how to switch from one device layer to another.

See Also

ActivateApDigi14 (APDIGI14)
ActivateApFossil (APFOSSIL)

ActivateApInt14 (APINT14)

Syntax

```
function ClassifyUart(BaseAddr : Word; CheckUart : Boolean) : UartType;
```

Purpose

Return the UartType for the specified UART.

Description

If CheckUart is True, ClassifyUart first checks that the chip at BaseAddr (the port address of the UART) really is a UART. If CheckUart is False, ClassifyUart skips this step (with nonsense results if the chip turns out not to be UART).

The possible UART types are:

- NoUart - returned only if CheckUart is True and the device is not a UART
- U8250B - a UART without a scratch register (installed in some original PCs)
- U8250A - a UART with a scratch register (could also be a 16450)
- U16550 - newer UART but without FIFO buffer
- U16550A - newer UART with 16 byte FIFO buffers for transmitting/receiving

APUART doesn't care what the UART is (as long as it's one of the above). Generally, you don't need to know either. There is one case where knowing the type of UART is important, however: the case where you want to use SetFifoBuffering. See "FIFO Buffering" earlier in this section.

Example

```
if ClassifyUart(BaseAddr, True) = U16550A then begin
  SetFifoBuffering(BaseAddr, NewFifoState, NewFifoLevel);
```

Turns FIFO buffering on if it is available.

See Also

SetFifoBuffering

Syntax

```
{ $IFDEF EventLogging }  
procedure DumpEvents(FName : PathStr);
```

Purpose

Write the EventQueue to a file.

Description

You can use this procedure to write an event logging report to disk in the file FName. This procedure also deallocates the circular event log and reprograms the system timer chip to its normal state.

If a report file cannot be created, then DumpEvents returns with ecEventFileError. If any I/O errors (like ecDiskFull) are encountered while writing the report, AsyncStatus is set to that error. Whether the report is written or not, event logging is always turned off.

See "Event Logging" earlier in this section for a description of the generated report.

See Also

InitEventLogging

Syntax

```
function FifoStatus(BaseAddr : Word) : Boolean;
```

Purpose

Return True if FIFO buffering is on for the UART at the specified port address.

Description

This function returns the status of the FIFO buffer on the UART at the specified port address (True if buffering is enabled, False if it's not enabled). It returns False if the UART doesn't support buffering. The results are unpredictable if the device at BaseAddr isn't a UART.

Unfortunately, it's not possible to return the trigger level of the receiver FIFO; we can determine only if it's on or off.

Note that enabling or disabling buffering always affects both transmit and receive buffers; you cannot change them independently. Currently, Async Professional takes advantage only of receive buffering.

Example

```
if ClassifyUart(BaseAddr, True) = U16550A then  
  if FifoStatus(BaseAddr) then  
    WriteLn('FIFO buffering is on')  
  else  
    WriteLn('FIFO buffering is off')
```

Displays the status of FIFO buffering on a U16550A UART.

See Also

ClassifyUart

SetFifoBuffering

InitEventLogging

Syntax

```
{IFDEF EventLogging}  
procedure InitEventLogging(Events : Word);
```

Purpose

Allocate an event buffer, reprogram and calibrate the system timer.

Description

This procedure starts the Async Professional EventLogging facility. It allocates an event buffer large enough to hold the requested number of events (setting AsyncStatus to ecOutOfMemory if the allocation fails). The largest acceptable value for Events is 10919 events.

This function also reprograms the system timer so that it can be read in a high resolution mode (for microsecond timing). This change in timer programming will not affect any other parts of your program. There may be a conflict, however, if some other part of your program also reprograms the system timer. In such cases, you may not be able to use event logging.

After calling this procedure, Async Professional starts building a circular log of all ISR events. The log can later be written to a file by calling DumpEvents.

To use this procedure, you must first turn on EventLogging in APDEFINE.INC and recompile APUART.

Example

```
var  
  ComPort : UartPort;  
begin  
  ComPort.Init(...);  
  InitEventLogging(1000);  
  {...use the port...}  
  DumpEvents('EVENT.LOG');  
  ComPort.Done(...);  
end;
```

After instantiating a port object, event logging is turned on with a buffer large enough to hold 1000 events. The event log is written to disk at the end of the program.

See Also

DumpEvents

Syntax

```
procedure RestoreUartState(BaseAddr : Word; PSR : PortSaveRec);
```

Purpose

Restore a communications chip to its previous state.

Description

This procedure attempts to restore the state of the UART at the specified port address from a state previously saved in PSR by SaveUartState.

We use the term "attempts" because not all of the registers can be restored. The following items are restored: the old baud rate, parity, data and stop bits settings, modem control values, and interrupt settings. The following items are not restored: a pending received character and the exact FIFO settings. If the FIFO was previously on, it is turned on again with a trigger level of 8 (which might be different from the old trigger level).

Example

```
var
  PSR : PortSaveRec;
...
  SaveUartState(DefBaseAddr [Com1], PSR);
  {...alter the state of the UART...}
  RestoreUartState(DefBaseAddr [Com1], PSR);
```

This saves the state of the Com1 port and later restores it.

See Also

SaveUartState

Syntax

```
procedure SaveUartState(BaseAddr : Word; var PSR : PortSaveRec);
```

Purpose

Save the state of the UART at the specified port address.

Description

This procedure attempts to save the current state of the UART at BaseAddr in PSR. See RestoreUartState for details on what can and cannot be saved.

Example

See the example for RestoreUartState.

See Also

RestoreUartState

SendLongBreak / SetFifoBuffering

Syntax

```
procedure SendLongBreak(BaseAddr : Word; Count : Byte);
```

Purpose

Send a long serial line break.

Description

This procedure sends a "long break" to the remote system. Count is the duration, in character-times, of the break. Thus, a Count value of two sends a break that is two character-times long. (See the glossary for a definition of character-time.)

Use this procedure for systems where the normal break routine, SendBreak, doesn't seem to be generating a break at the remote system. A normal break is only one character-time long and some mini-computer and mainframe systems need a break condition somewhat longer than this.

You'll need to experiment with Count values to find a value that works with any given remote system. Note that SendLongBreak leaves interrupts off while sending the break so you should use the smallest value possible here. Generally, you should start testing with a value of two and increment as required to generate the break at the remote system.

Example

```
BaseAddr := DefBaseAddr[Com1];  
SendLongBreak(BaseAddr, 2);
```

Sends a long break using Com1.

Syntax

```
procedure SetFifoBuffering(BaseAddr : Word; Enable : Boolean; Level : Byte);
```

Purpose

Turns FIFO buffering on/off for the UART at the specified port address.

Description

This procedure turns FIFO buffering on (Enable = True) or off (Enable = False) for the UART at BaseAddr.

Level is the trigger level for the FIFO to use to signal receive interrupts. The only acceptable values are 1, 4, 8 or 14. This value is not limit checked. For incorrect levels, SetFifoBuffering just uses the next lower legitimate value.

Your program should call ClassifyUart after initializing the port and make subsequent calls to SetFifoBuffering only if the UART is known to have a FIFO buffer. Note that calling SetFifoBuffering for a non-buffered UART will not cause any harm, but it does waste time.

See "FIFO Buffering" earlier in this section for more information.

Example

```
BaseAddr := DefBaseAddr[Com1];  
if ClassifyUart(BaseAddr, True) = U16550A then begin  
  SetFifoBuffering(BaseAddr, True, 14);  
  WriteLn('FIFO buffering activated');  
end else  
  WriteLn('COM1 is not a 16550A');
```

Enables FIFO buffering for Com1 with a trigger level of 14, if possible, or displays an error message if it isn't.

See Also

ClassifyUart

FifoStatus

Syntax

```
function UartTest1(BaseAddr : Word) : Boolean;
```

Purpose

Check for a UART by reading the interrupt enable register, interrupt ID register, and modem control register.

Description

This function checks to see if the device at the specified port address really is a UART. The test is a passive read of what should be the interrupt enable register (IEReg), interrupt ID register (IIDreg), and the modem control register (MCreg). Each of these registers has bits that are never set. If any of them are set, this function returns False.

Obviously, this test isn't foolproof, but when used with the other UART tests, it should reliably detect a UART.

Some words about UART detection are in order. The intent was to provide a passive test (just reading some registers) and follow that with an active test (change a register to see if it behaves like a UART). When the device is actually a UART, all this works quite well. The danger is that the device at BaseAddr might *not* be a UART. In those cases, it would be nice to make sure that these tests don't disturb that device, whatever it might be.

Unfortunately, this proves to be impossible. That is, some devices are disturbed merely by reading their registers. One case of this occurs with ArcNet network boards. The default address of these boards is \$2E0-\$2EF, which happens to overlap the default address space of Com4. Just reading any of these registers disrupts the normal operation of that network board (crashing the workstation's network shell).

The only foolproof solution is to rely on UartTest3 (which returns True only if BaseAddr is in the BIOS data area). The problem here is that the power-on routines of some machines don't look for ports beyond Com2. Where this is a problem, you can run the public domain program BIOSFIX (it is supplied in the \BONUS subdirectory) *early* in your AUTOEXEC.BAT routine (before any network or other add-in board drivers are loaded). At that time it's safe to use UartTest1 and UartTest2—BIOSFIX does so and makes sure all com port addresses are added to the BIOS data area (up to a total of four).

Example

```
{Make sure there is a UART at this address}
BaseAddr := DefBaseAddr[Com1];
if not UartTest1(BaseAddr) or not UartTest2(BaseAddr) then
  WriteLn('No UART found');
```

Checks for a UART at the address for Com1.

See Also

UartTest2

UartTest3

UartTest2 / UartTest3

Syntax

```
function UartTest2(BaseAddr : Word) : Boolean;
```

Purpose

Check for a UART by writing to the interrupt enable register.

Description

This function checks to see if the device at the specified port address really is a UART. This is an active test that actually writes to what should be the interrupt enable register. It saves the old contents before doing so and restores the contents when finished.

The only possible danger is that the device is *not* a UART and writing at this address may affect whatever this device really is. However, this is safer (but not foolproof) if you precede it with UartTest1.

Example

See the example for UartTest1.

See Also

UartTest1

Syntax

```
function UartTest3(BaseAddr : Word) : Boolean;
```

Purpose

Check for a UART by reading the serial port data portion of the BIOS data area.

Description

This function checks for BaseAddr in the serial port data portion of the BIOS data area. This part of the BIOS data area is initialized at bootstrap. Assuming that the BIOS properly detected the UARTs in your system, you will find those base addresses here.

Unfortunately, the BIOS looks only at the standard UART addresses and some BIOS's look only for Com1 and Com2. In any case, checking to see if the BIOS knows about a UART might be a nice double-check.

Example

```
if UartTest3(DefBaseAddr[Com1]) then  
  WriteLn('Known to BIOS')  
else  
  WriteLn('Not known to BIOS');
```

Checks to see if the BIOS detected a UART at the address for Com1.

See Also

UartTest1

D. Fossil Device Layer: APFOSSIL

FOSSIL is short for "Fido Opus SEAdog Standard Interface Layer." FOSSIL drivers are DOS device drivers or TSRs that are loaded before your communications software and manage all of the low-level details of serial communication. Your programs communicate with the FOSSIL driver through Int \$14. FOSSIL drivers are used by some bulletin board systems and utilities.

APFOSSIL provides the connection between your application and a FOSSIL driver. The APFOSSIL module contains the core functions necessary to implement a FOSSIL device layer. By taking advantage of the Async Professional layered architecture, this FOSSIL device layer transparently replaces the built-in UART support. When using the FOSSIL device layer, all components of Async Professional from the interface layer on up (including the modem support, protocols, and so on) make calls to a FOSSIL driver rather than the built-in UART driver. This means, of course, that the host machine must have a FOSSIL driver loaded. The term "FOSSIL driver" refers to the FOSSIL device driver or TSR that you must load before running any communications programs. The term "FOSSIL device layer" refers to functions provided by APFOSSIL.PAS.

Since APFOSSIL is a device layer module, its main routines are documented in §4.G (Core Routines). This section describes only the APFOSSIL specific routines.

FTEST and FTESTO are programs that demonstrate how to use the APFOSSIL functions. See the source code for FTEST/FTESTO for more information. COMTEST and COMTESTO demonstrate how to use APFOSSIL together with other device layers. See §1.E for more information on COMTEST/COMTESTO.

FOSSIL Device Driver

You must already have a FOSSIL driver to use the APFOSSIL device layer. If you don't have a FOSSIL driver, you can download one from the TurboPower BBS (the telephone number is listed on the title page of this manual). The two drivers that are available are X00nnn.EXE and BNUnnn.ZIP (where nnn is the current version number).

The Async Professional FOSSIL device layer was tested with the two of the most popular FOSSIL drivers: BNU version 1.70 and X00 version 1.24. Presumably, this device layer will work with any other well-behaved FOSSIL driver, but this is not guaranteed.

The feature set provided by these standard FOSSIL drivers is smaller than the feature set provided by the standard APUART device layer. What this means to you is that some parts of the Async Professional interface layer might not work the way they work when using the APUART device layer.

These differences are handled in two different ways. In some cases, reasonable guesses or reasonable initialized values are returned. For example, the APUART unit allows you to open a port without changing the current line parameters. You can then call GetLine to find out what line parameters you inherited. The FOSSIL driver also allows you to open a port without changing the line parameters, but it doesn't offer any way to find out what line parameters you've inherited (except for the baud rate). So what should GetLine return in this case for parity, databits and stopbits? In this and the majority of similar cases, initialized, but not necessarily correct, values (usually zero) are returned.

In cases that are more critical to proper program operation, an error is returned. For example, APUART allows you to request hardware flow control using DTR/DSR or RTS/CTS. But, the FOSSIL interface only supports RTS/CTS flow control. If you request DTR/DSR flow control when using the FOSSIL device layer, an `ecInvalidArgument` error is returned.

Once you understand how these issues affect the device and interface layers, you'll have a good feel for how they might impact the way you use these routines. Note that these issues affect only the interface layer of Async Professional. The higher layers, notably the protocol layer, work just fine with the FOSSIL device layer because they either anticipate and work around the FOSSIL differences, or the FOSSIL differences don't matter for what the protocol is doing. The notable differences in the FOSSIL driver are:

Line parameters - the FOSSIL driver cannot tell you the line parameters, except for baud rate, of a port you have just opened.

Line errors - the FOSSIL driver does not report any type of line errors except input buffer overrun.

Modem control - the FOSSIL driver cannot tell the state of DTR or RTS signals.

Modem status - the FOSSIL driver won't return any modem status information except data carrier detect (DCD).

Flow control - the FOSSIL driver only supports RTS/CTS hardware flow control, and it can't return the current state of flow control other than on/off (i.e., it can't report whether flow control is currently clear or blocked).

Declarations

Constants

```
DefFossilOptionsSimple = ptHandleFossilBug + ptPutCharWait;  
DefFossilOptions : Word = DefFossilOptionsSimple;
```

The default FOSSIL device layer options.

```
ptHandleFossilBug = $8000;  
ptTrueOutBuffFree = $4000;  
ptPutCharWait = $2000;
```

Port options that are used for Fossil ports only. See the §4.B, APPORT, for descriptions of the port options that are used for all device layers.

Some FOSSIL drivers have a bug where OutBuffUsed sometimes does not drop to zero when the output buffer is empty (the driver information field diOutUsed is one when it should be zero). When ptHandleFossilBug is set, APFOSSIL automatically sets OutBuffUsed to zero whenever it returns one. You should be aware of this behavior whenever checking for empty output buffer values in your program. Note that X00 has the bug, BNU does not. ptHandleFossilBug is on by default since this behavior is not harmful, even with a driver that does not have the bug.

ptTrueOutBuffFree tells the FOSSIL device layer whether the function OutBuffFree should return the actual amount of free space in the output buffer, or an indication of whether more data should be sent to the FOSSIL driver. If ptTrueOutBuffFree is on, OutBuffFree returns the actual amount of free space in the FOSSIL output buffer. If ptTrueOutBuffFree is off (as it is by default), OutBuffFree returns either 0 (if the output buffer is less than 90% free, meaning don't send any more data to the FOSSIL driver) or 65535 (if the output buffer is at least 90% free, meaning it is OK to send more data to the FOSSIL driver).

ptTrueOutBuffFree is off by default because most FOSSIL users tend to use small (256 or 512 byte) FOSSIL buffers. These small buffers can cause a problem because the Async Professional file transfer protocols wait until OutBuffFree returns a value large enough to hold an entire block. Since a block could be as large as 2078 bytes, the protocol might wait forever. You can turn ptTrueOutBuffFree on if you use FOSSIL buffers of 2078 or greater. Then the actual amount of free space will eventually get large enough for the largest possible block.

ptPutCharWait tells the FOSSIL device layer to use "wait" transmits when transmitting data. This option is on by default since most FOSSIL users use small FOSSIL buffers (256 or 512 bytes)

and it is possible for file transfer protocols to overflow such small buffers. If you are using large FOSSIL buffers, you can turn this option off and `ptTrueOutBuffFree` on.

Note that using the default options (`ptTrueOutBuffFree` not set, `ptPutCharWait` set) may cause problems when trying to use the file transfer protocols in background mode since the FOSSIL driver will sometimes wait to transmit a block of data. If you want to use background file transfers, you *must* provide a FOSSIL output buffer of at least 2078 bytes and you must turn off `ptPutCharWait`.

ValidLineStatus : Byte = \$03;

Mask for valid line status bits. Most FOSSIL drivers return only bit 0 (data ready) and bit 1 (overflow). Due to this shortcoming, the following interface layer routines are not accurate for the APFOSSIL device layer: `CheckLineError`, `CheckLineBreak`, `CheckTE`, `CheckTHRE`, and `CheckFIFOError`. If you are confident that your FOSSIL driver reliably returns other line status bits, you can adjust `ValidLineStatus` to account for those bits.

ValidModemStatus : Byte = \$80;

Mask for valid modem status bits. Most FOSSIL drivers return only bit 7 (Data Carrier Detect). Due to this shortcoming, the following interface layer routines are not accurate for the APFOSSIL device layer: `CheckCTS`, `CheckDSR`, `CheckRI`, `CheckDeltaCTS`, `CheckDeltaDSR`, `CheckDeltaRI`, and `CheckDeltaDCD`. If you are confident that your FOSSIL driver reliably returns other line status bits, you can adjust `ValidModemStatus` to account for those bits.

ActivateApFossil

Syntax

procedure ActivateApFossil;

Purpose

Register APFOSSIL as the active device layer.

Description

You can have almost any number of device layer modules in your program, but only one of them can be active at any one time. This routine makes APFOSSIL the active device layer module.

This routine is called automatically by the initialization block of APFOSSIL if AutoDeviceInit is defined in APDEFINE.INC. If your program uses multiple device layer modules, then you should call ActivateApFossil to insure that APFOSSIL is the active one. You would also call this if your application supports changing the device layer module at run time.

See "Async Professional Architecture" in §2.C for more details on device layer modules.

See Also

ActivateApDigi14 (APDIGI14)

ActivateApInt14 (APINT14)

ActivateApUart (APUART)

E. DigiBoard Device Layer: APDIGI14

DigiBoard is a manufacturer of multiport boards for serial communication. Most of these boards are "intelligent" in that they handle the majority of the low-level communications work. Async Professional communicates with these boards by making Int \$14 calls to the DigiBoard device driver.

APDIGI14 provides the connection between your application and a DigiBoard. The APDIGI14 unit contains the functions that implement a Digi14 device layer. By taking advantage of the Async Professional layered architecture, this Digi14 device layer transparently replaces the built-in UART support. When using the Digi14 device layer, all components of Async Professional from the interface layer on up (including the modem support, protocols, and so on) make calls to a Digi14 driver rather than the built-in UART driver. This means, of course, that the host machine must have a Digi14 driver loaded and be using a DigiBoard multiport board that supports the DigiBoard Int \$14 universal device driver. The term "Digi14 driver" refers to the DigiBoard universal device driver (XIDOS5.SYS) that you must load before running any communications programs. The term "Digi14 device layer" refers to functions provided by APDIGI14.PAS.

Since APDIGI14 is a device layer module, its main routines are documented in §4.G (Core Routines). This section describes only the APDIGI14 specific routines.

DTEST and DTESTO are programs that demonstrate how to use the APDIGI14 routines. See the source code for DTEST/DTESTO for more information. COMTEST and COMTESTO demonstrate how to use APDIGI14 together with other device layers. See §1.E for more information on COMTEST/COMTESTO.

DigiBoard Universal Device Driver

The "universal device driver" is DigiBoard's name for their interrupt \$14-based device driver that works with any of the following DigiBoards: COM/Xi, PC/Xi, PC/Xe and, presumably, any future DigiBoards. This device driver replaces the board-specific device driver that used to accompany each board. By using universal device driver conventions, this Digi14 device layer will work with any of the above mentioned DigiBoard products.

For this to work, however, you are responsible for properly configuring your DigiBoard multiport board, properly configuring XIDOS5.SYS, and installing XIDOS5.SYS in your CONFIG.SYS file. Refer to your DigiBoard instructions for further information.

The Async Professional Digi14 device layer was tested with version 4.05 of the universal device driver, which was the most recent version at the time of our testing.

The feature set provided by the Digi14 device driver is slightly smaller than the feature set provided by the standard APUART device layer. What this means to you is that some parts of the Async Professional interface layer might not work the way they work when using the APUART device layer.

These differences are handled by returning a reasonable guess or returning less information than APUART might return. For example, when software flow control is in use, APUART provides detailed information about whether flow control is currently clear, blocked by the remote, or blocked by the local machine. The Digi14 driver can only determine whether flow control is currently on or off—not whether it is blocked or clear. So, the only values returned by SWFlowState are fsOff or fsClear (fsClear means flow control is on, but could be either clear or blocked).

Input Buffer Usage

Because the DigiBoard driver stores information bytes as well as data bytes in the input buffer, it must scan the entire input buffer in response to calls to InBuffUsed and InBuffFree. The time required for this search is trivial when the buffer is nearly empty, but can grow to 0.5 milliseconds when the buffer has more than 1 or 2KB of data (times will vary with hardware).

You should avoid calling InBuffUsed or InBuffFree in time-critical loops. If this can't be avoided, you can supply your own InBuffUsed and InBuffFree routines that use DigiBoard function \$15 instead \$0A. Function \$15 is faster than function \$0A, but, it returns the number of total bytes in the input buffer instead of the number of characters. This means that InBuffUsed will probably return a number higher than the actual number of characters in the input buffer.

DigiBoard Non-Intelligent Boards

Although this subsection focuses on DigiBoard intelligent boards, Async Professional also works with DigiBoard non-intelligent boards (also called dumb boards). When using a non-intelligent board, use the APUART device layer unit, not APDIGI14.

Since these boards share IRQs between ports, you must enable the ShareIrq equate in APUART.ASM by removing the semicolon in front of it. Rebuild the library using 'MAKE -fAPRO' or 'MAKE -fAPROO'. You *must* use one of these supplied make files so that all of the different versions of APUART?.OBJ are created. Next, set up the UART addresses and IRQ via the DigiBoard COM.SYS device driver and inform Async Professional of these addresses by calling SetUart before opening each port.

Declarations

Constants

```
DefDigi14OptionsSimple = 0;  
DefDigi14Options : Word = DefDigi14OptionsSimple;
```

The default Digi14 device layer options (none).

```
ptReadWriteWait      = $8000;  
ptClearFlow          = $4000;
```

Port options that are used for DigiBoard ports only. See §4.B, APPORT, for descriptions of the port options that are used for all device layers.

ptReadWriteWait, when set, causes the PutChar/GetChar routines to wait two seconds for room in the transmit buffer or wait two seconds for a character to receive. Generally, there's no reason to wait in either case, so this option is not set by default.

ptClearFlow, when set, causes the default flow control to be disabled when a Digi14 port is opened. This may be necessary if flow control was enabled when the DigiBoard XIDOS5.SYS driver was configured by the DigiBoard configuration program XIDOSCFG.EXE. If flow control was turned on in XIDOS5.SYS, then flow control is in operation. However, the Digi14 device layer doesn't know that flow control is on and it always reports that it is off. Additionally, while hardware flow control is on, the Digi14 device driver ignores all requests to set or reset the DTR/RTS lines. If you want to control these lines yourself, or you want to set and monitor flow control yourself, then you should set the ptClearFlow option.

We recommend turning on ptClearFlow since it gives you the most flexibility and visibility over the hardware control lines and flow control in general. We suspect, however, that most DigiBoard users are used to setting flow control in the driver and prefer that the program not interfere with those settings. For this reason ptClearFlow is off by default.

Syntax

procedure ActivateApDigi14;

Purpose

Register APDIGI14 as the active device layer.

Description

You can have almost any number of device layer modules in your program, but only one of them can be active at any one time. This routine makes APDIGI14 the active device layer module.

This routine is called automatically by the initialization block of APDIGI14 if AutoDeviceInit is defined in APDEFINE.INC. If your program uses multiple device layer modules, then you should call ActivateApDigi14 to insure that APDIGI14 is the active one. You would also call this if your application supports changing the device layer module at run time.

See "Async Professional Architecture" in §2.C for more details on device layer modules.

See Also

ActivateApFossil (APFOSSIL)
ActivateApInt14 (APINT14)

ActivateApUart (APUART)

F. BIOS Interrupt \$14 Device Layer: APINT14

APINT14 is a device layer unit that relies only on the standard PC BIOS interrupt \$14 services for accessing serial ports. That means that this unit offers no performance or features beyond the standard BIOS interrupt \$14 services, which are widely recognized as being rather lame.

Nevertheless, APINT14 provides several useful functions. First, it's a demonstration of an alternative device layer. More than that, it's a demonstration of an *unbuffered* device layer. Generally, whether a device layer is buffered or not won't really matter to your application program. It does matter, though, to the interface layer, which must work with both buffered and unbuffered device layers. APINT14 was the test bench for this and may also prove a useful example if you happen to be writing a device layer for another unbuffered device.

Second, it might prove useful for some off-the-wall clones that don't use standard UARTs, but do honor the interrupt \$14 services. The likelihood of this is rather low. Even if it were the case, the performance and abilities of applications built on this device layer would be rather limited. However, should such a situation arise, APINT14 might work where nothing else can.

And third, APINT14 provides a basis for device layers that use a derivative of the Int \$14 calling interface. APFOSSIL and APDIGI14 are examples of such Int \$14 devices. Many other devices (such as network shared modem pools) use an Int \$14 calling interface. APINT14 could be used as a basis if you need to develop this type of device layer.

Since APINT14 is a device layer unit, its routines are documented in §4.G (Core Routines). APINT14 interfaces only one other routine, which is discussed in this section.

Syntax

```
procedure ActivateApInt14;
```

Purpose

Register APINT14 as the active device layer.

Description

You can have almost any number of device layer units in your program, but only one of them can be active at any one time. This procedure makes APINT14 the active device layer unit.

This procedure is called automatically by APINT14's initialization block if AutoDeviceInit is defined in APDEFINE.INC. If your program uses multiple device layer units, you should call ActivateApInt14 to ensure that APINT14 is the active one.

You would also call this if your application supports changing the device layer unit at run time.

See "Async Professional Architecture" in §2.C for more details on device layer units.

Example

```
ActivateApUart; {make the APUART unit active}  
{... use other Async Professional calls ...}  
ActivateApInt14; {make the APINT14 unit active}
```

Shows how to switch from one device layer to another.

See Also

ActivateApDigi14 (APDIGI14)

ActivateApFossil (APFOSSIL)

ActivateApUart (APUART)

G. Core Routines

The core routines are those basic routines (like GetChar and PutChar) that all device layers must support. The core routines are defined at the abstract layer (APPORT), and implemented in a device layer (APUART, APFOSSIL, APDIGI14, and APINT14). All interface layer routines (APCOM and OOCOM) build upon these core routines.

While this architecture provides some tremendous advantages in supporting future serial devices, it also provides some challenges in presenting this documentation in a sensible format. A typical program written using Async Professional will freely mix routines from the device layer (documented here) and routines from the interface layer (documented in Chapter 5). The best we can do is alert you to this arrangement and suggest that, in time, you'll know intuitively whether a routine is a core routine or an interface routine (and, therefore, know which chapter to turn to when looking for information).

Remember that you also have tools like POPHELP and the alphabetic index of routines at the end of the manual.

Syntax

```
procedure ActivatePort(P : PortRecPtr; Restore : Boolean);
```

Purpose

Turn on interrupts for this port.

Description

This procedure turns interrupts on for the specified port. If the Restore parameter is True, it also reinstalls the interrupt handler (if any) used by the device layer to service the port.

The typical application for ActivatePort is in a program that uses a swapping DOS Exec function to execute other programs. In such cases, it is always used in conjunction with SavePort, RestorePort, and DeactivatePort, and the Restore parameter is always True.

APUART requires this procedure to reactivate the device layer after being shut down (for example, after a swapping DOS Exec).

APFOSSIL, APDIGI14 and APINT14 do not require this call and simply ignore it.

Example

```
SavePort(P, PSR);  
DeactivatePort(P, True);  
{... execute another program ...}  
RestorePort(P, PSR);  
ActivatePort(P, True);
```

Save the current state of the port and deactivate it prior to executing another program. On return, restore the state of the port and activate it.

See Also

DeactivatePort
RestorePort

SavePort

Syntax

```
procedure BufferFlush(P : PortRecPtr; FlushIn, FlushOut: Boolean);
```

Purpose

Flush the input and/or output buffers.

Description

This routine flushes (discards the contents of) the input and/or output buffers. It is intended primarily for internal use. You should use the FlushInBuffer/FlushOutBuffer routines in APCOM/OOCOM.

If FlushIn is True, the input buffer is flushed. If FlushOut is True, the output buffer is flushed. Both buffers can be flushed at the same time.

APINT14 doesn't use buffers and returns ecNotSupported if this function is called.

See Also

FlushInBuffer (APCOM)
AbstractPort.FlushInBuffer

FlushOutBuffer (APCOM)
AbstractPort.FlushOutBuffer

BufferStatus / CharReady

Syntax

```
procedure BufferStatus(P : PortRecPtr;  
                      var InFree, OutFree, InUsed, OutUsed : Word);
```

Purpose

Return buffer usage information.

Description

This routine returns complete information about current input and output buffer usage. This is intended primarily for internal use; you should generally use the InBuffFree, OutBuffFree, InBuffUsed and OutBuffUsed routines in APCOM/OOCOM. However, you may find calling BufferStatus more convenient if you need to get all buffer status information at once.

APINT14 doesn't use buffers. It returns zero for InUsed and OutUsed and 65535 for InFree and OutFree.

APFOSSIL has several special behaviors when returning buffer information. See the FossilPort options in the APFOSSIL Declarations in §4.D for a complete discussion of these issues.

See Also

InBuffFree (APCOM)
AbstractPort.InBuffFree
InBuffUsed (APCOM)
AbstractPort.InBuffUsed

OutBuffFree (APCOM)
AbstractPort.OutBuffFree
OutBuffUsed (APCOM)
AbstractPort.OutBuffUsed

Syntax

```
function CharReady(P : PortRecPtr) : Boolean;
```

Purpose

Return True if at least one character is ready for retrieval.

Description

This function returns True if at least one character is ready to be retrieved from the specified port.

Example

```
if CharReady(P) then begin  
  GetChar(P, C);  
  Write(C);  
end;
```

If a character is ready, get it and display it on the screen.

See Also

GetChar

PeekChar

Syntax

```
procedure DeactivatePort(P : PortRecPtr; Restore : Boolean);
```

Purpose

Turn off interrupts for this port.

Description

This procedure turns interrupts off for the specified port. Of course, this means that any incoming characters are lost while the interrupts are off. If the Restore parameter is True, it also restores any interrupt vector grabbed by the device layer to service the port.

The typical application for DeactivatePort is in a program that uses a swapping DOS Exec function to execute other programs. In such cases, it is always used in conjunction with SavePort, RestorePort, and ActivatePort and the Restore parameter is always True. When using a non-swapping Exec function, it is not necessary to deactivate the port (SavePort and RestorePort are sufficient).

APUART requires this procedure to deactivate the device layer before a swapping DOS Exec. APFOSSIL, APDIGI14, and APINT14 do not require this call and simply ignore it.

Example

See the example for ActivatePort.

See Also

ActivatePort
RestorePort

SavePort

Syntax

```
procedure DonePort(var P : PortRecPtr);
```

Purpose

Close the specified port.

Description

This procedure closes the specified port. P must be a pointer to a previously opened port. On return, P is set to Nil.

In the case of APUART, this function turns off all UART interrupt conditions, disables interrupts for this vector, and restores the old interrupt vector.

For APFOSSIL, this function makes the appropriate call to the FOSSIL driver to close the port.

For APDIGI14, this function makes the appropriate call to the DigiBoard driver to close the port.

For APINT14, DonePort doesn't really do anything except dispose of the PortRec pointed to by P, since APINT14 doesn't use interrupt driven communications.

Example

```
InitPort(P, Com1, 1200, NoParity, 8, 1, 500, 500, DefPortOptions);
```

```
...
```

```
DonePort(P);
```

Opens a port and later closes it.

```
UP.InitCustom(Com1, 1200, NoParity, 8, 1, 500, 500, DefPortOptions);
```

```
...
```

```
UP.Done;
```

The OOP version of opening a port and later closing it.

See Also

InitPort

GetChar / GetLine

Syntax

```
procedure GetChar(P : PortRecPtr; var C : Char);
```

Purpose

Return the next character from the specified port.

Description

This procedure returns the next character waiting to be retrieved from the specified port in C. It is a no-wait call; it returns immediately whether a character is ready or not. If a character is ready, that character is returned in C and AsyncStatus is set to ecOk. If a character is not ready, C is set to \$FF and AsyncStatus is set to ecBufferIsEmpty (whether input is actually buffered or not).

Example

See the example for CharReady.

See Also

CharReady

PeekChar

Syntax

```
procedure GetLine(P : PortRecPtr; var Baud : LongInt;  
                  var Parity : ParityType; var DataBits : DataBitType;  
                  var StopBits : StopBitType; FromHardware : Boolean);
```

Purpose

Get the line parameters directly from the hardware, if possible.

Description

If FromHardware is False, the hardware is not accessed directly. The function simply returns the values stored at the location pointed to by P.

If FromHardware is True, the device layers do the following:

APUART and APDIGI14 return the current settings for baud rate, parity, and data and stop bits for the specified port.

APFOSSIL returns an accurate baud rate only, the other line parameters are always returned as NoParity, 8 data bits and 1 stop bit.

APINT14 returns no parameters at all and generates an ecNotSupported error.

Example

```
InitPort(P, Com1, 1200, NoParity, 8, 1, 500, 500, DefPortOptions);
```

```
...
```

```
GetLine(P, Baud, Parity, DataBits, StopBits, True);
```

In this case, Baud is 1200, Parity is NoParity, DataBits is 8, and StopBits is 1.

Syntax

```
procedure GetModem(P : PortRecPtr; var DTR, RTS : Boolean);
```

Purpose

Get the DTR and RTS settings directly from the port.

Description

This returns the state of port P's modem control signals. The DTR (Data Terminal Ready) signal generally means that this com port is ready and working. The RTS (Request To Send) signal generally means this com port is ready to receive data; this is the standard hardware flow control signal. See "Flow Control" in §5.C for more information. Other computers, modems, lab equipment, etc. might attach different meanings to these signals.

APFOSSIL and APINT14 do not support this function and generate an `ecNotSupported` error.

Example

```
const
  OnOff : array[Boolean] of String[3] = ('Off', 'On ');
...
GetModem(P, DTR, RTS);
WriteLn('DTR is ', OnOff[DTR], ' RTS is ', OnOff[RTS]);
```

Display the current state of the DTR and RTS signals.

See Also

SetModem

Syntax

```
procedure GotError(P : PortRecPtr; StatusCode : Word);
```

Purpose

Report an error to the user defined error handler, if one exists.

Description

This procedure is called internally when an error occurs. If an error handler was installed by calling `SetErrorProc`, `GotError` passes the error code (`StatusCode`) on to it. Typically, an application will not call `GotError` directly. It might, however, if you were implementing your own device layer or your own file transfer protocol, for example.

No default error handler is supplied. If you don't supply one, the internal calls to `GotError` do nothing.

Example

```
GotError(P, epFatal+ecInsufficientMemory);
Exit;
```

Report an out of memory error and exit.

See Also

AbstractPort.SetErrorProc

SetErrorProc (APCOM)

HWFlowGet / HWFlowSet

Syntax

```
($IFDEF UseHWFlow)
function HWFlowGet(P : PortRecPtr) : FlowState;
```

Purpose

Return the state of hardware flow control.

Description

This routine returns one of the following states:

- fsOff - flow control is off
- fsClear - flow control is on but clear
- fsTransWait - the transmitter is blocked
- fsRecWait - the receiver is blocked
- fsAllWait - both transmitter and receive are blocked

This routine is intended primarily for internal use. You should use the HWFlowState function in APCOM/OOCOM.

APINT14 always returns fsOff and the error ecNotSupported.

APFOSSIL and APDIGI14 return only fsOff or fsClear. They can report whether flow control is on, but no details beyond that.

See Also

HWFlowSet

HWFlowState (APCOM)

AbstractPort.HWFlowState

Syntax

```
($IFDEF UseHWFlow)
procedure HWFlowSet(P : PortRecPtr; Enable : Boolean;
  BufferFull, BufferResume : Word; Options : Word);
```

Purpose

Start or stop hardware flow control.

Description

This routine starts or stops hardware flow control for the current device layer. It is intended primarily for internal use. You should use the HWFlowEnable/HWFlowDisable routines in APCOM/OOCOM.

APUART supports all hardware flow control options described in §5.C.

APFOSSIL supports only the option combination hfUseRTS or hfRequireCTS. All other option combinations return ecInvalidArgument. Additionally, the BufferFull and BufferResume parameters are ignored since the FOSSIL driver makes its own decision about the buffer full and resume points.

APDIGI14 supports all hardware flow control options except the inverted signal options (hfDTRActiveLow..hfCTSActiveLow). These return an ecInvalidArgument error.

APINT14 doesn't support hardware flow control and returns ecNotSupported.

See Also

HWFlowDisable (APCOM)

AbstractPort.HWFlowDisable

HWFlowEnable (APCOM)

AbstractPort.HWFlowEnable
HWFlowGet

Syntax

```

procedure InitPort(var P : PortRecPtr; ComName : ComNameType; Baud : LongInt;
  Parity : ParityType; DataBits : DataBitType;
  StopBits : StopBitType; InSize, OutSize : Word;
  Options : Word);

```

Purpose

Open the specified port.

Description

This procedure opens the com port specified by ComName (Com1 through Com36) and, if successful, returns a pointer (P) to a corresponding PortRec record.

Baud is the baud rate for this port (50 to 115200). Parity is the parity setting for this port (NoParity through SpaceParity). DataBits is the number of data bits in a byte (5 through 8). StopBits is the number of stop bits after each byte (1 or 2).

The Options parameter will typically be DefPortOptions (see §4.B, APPORT, for a complete list of the port options used by all device layers and see each device layer for specialized port options). Note that the state of the ptRaiseModemOnOpen option is of particular relevance for InitPort, since it specifies the initial treatment of this machine's DTR (Data Terminal Ready) and RTS (Request To Send) signals. If the option is on, both of these signals are raised. If it is off, neither of these signals is raised. If you need to raise only one of these, then you should call SetModem after the com port is opened.

The following errors can occur during a call to InitPort:

Error	Meaning
ecOutOfRange	InSize or OutSize was out of range
ecOutOfMemory	Not enough heap space for PortRec or buffer
ecNotaUart	No UART at the default base address (APUART only)
ecNoMorePorts	No more active ports allowed (need to close a port)
ecInvalidBaudRate	Baud rate is out of range for the current device
ecBadPortNumber	ComName > Com4 and machine is not a PS/2 (APINT14 only)
ecInvalidParity	Parity is MarkParity or SpaceParity (APINT14 only)

In APUART two buffers are allocated: an input buffer of InSize bytes, and an output buffer of OutSize bytes. InSize and OutSize should both be in the range 10 through 65521.

APFOSSIL ignores the InSize and OutSize parameters because the FOSSIL driver uses its own buffers (set when the driver was installed). APFOSSIL has a maximum baud rate of 19200 baud.

APDIGI14 ignores the InSize and OutSize parameters because the buffers are contained in DigiBoard memory.

APINT14 ignores the InSize and OutSize parameters because it doesn't use buffers. Parity settings of MarkParity or SpaceParity are not allowed by APINT14. APINT14 has a maximum baud rate of 9600 baud.

Example

```

InitPort(P, Com1, 9600, NoParity, 8, 1, 2048, 2078, DefPortOptions);
if AsyncStatus <> ecOk then
  WriteLn('Failed to open port, Status = ', AsyncStatus);

```

Open Com1 at 9600 baud with no parity, 8 data bits, and 1 stop bit. Allocate an input buffer of 2048 bytes and an output buffer of 2078 bytes. This does the same thing as 'InitPortFast(P, Com1, 9600);'.

See Also

DonePort
InitPortFast (APCOM)

InitPortKeep
SetModem

InitPortKeep / PeekChar

Syntax

```
procedure InitPortKeep(var P : PortRecPtr; ComName : ComNameType;  
    InSize, OutSize : Word);
```

Purpose

Open the specified port without changing the current line parameters.

Description

This procedure can be used instead of InitPort when you do not want to change the current line settings (baud rate, parity, data bits, and stop bits).

One common application for InitPortKeep is when you are writing a program, intended to be executed by a communications program, that implements a file transfer protocol not supported by the communications program. In such a case, your program must use the line settings established by the calling program.

InitPortKeep can generate the same errors as InitPort, with the exception of ecInvalidParity.

APFOSSIL, APDIGI14, and APINT14 ignore the InSize and OutSize parameters.

Example

```
InitPortKeep(P, Com1, 500, 500);  
{...check for errors...}  
GetLine(P, Baud, Parity, DataBits, StopBits, True);
```

This initializes P for Com1 using the existing line settings.

See Also

InitPort

Syntax

```
procedure PeekChar(P : PortRecPtr; var C : Char; PeekAhead : Word);
```

Purpose

Look at characters in the input buffer without actually removing them.

Description

This procedure allows you to look ahead in the input buffer without actually extracting the character (i.e., the character is returned but the buffer pointers are not changed).

The value for PeekAhead must be less than or equal to the current number of characters in the buffer. If you try to peek at a character that's not in the buffer, AsyncStatus reports ecBufferIsEmpty. It might not necessarily be empty; but it contains fewer characters than the value of PeekAhead.

PeekChar does *not* return any line errors (parity, framing or overrun errors).

APDIGI14 can only peek ahead one character. Attempts to peek ahead more than one character generate an ecInvalidArgument error.

APINT14 and APFOSSIL cannot peek ahead, so they generate an ecNotSupported error.

Example

```
PeekChar(P, C, 1);
```

Returns the next character in the input buffer, if any, in C.

See Also

CharReady

GetChar

Syntax

```
procedure PutChar(P : PortRecPtr; C : Char);
```

Purpose

Transmit a character.

Description

This procedure transmits character C either by adding it to the output buffer or by sending it directly to the port. This is a no-wait I/O procedure; it returns immediately whether it succeeds or not.

Once a character is placed in the buffer, you have no knowledge of exactly when that character will be sent to the remote device. If no flow control is in place, characters in the buffer are taken from the buffer and transmitted at about the one character-time rate. That is, every time the port finishes shifting a character out, it immediately asks for another one. The only gap between the characters is the time it takes the kernel to extract another character from the buffer and pass it to the port (just a couple of dozen assembly language instructions).

If you are using hardware or software flow control (see §5.C for more information on flow control), you have even less control over when a character is actually transmitted. At any time, the remote may ask the kernel to stop transmitting. Once stopped, the Async Professional kernel waits (possibly forever) until the remote says it's OK to continue.

If the device layer is APUART, the character is added to the transmit buffer. If there is no room in the buffer, it reports an `ecBufferIsFull` error.

For more details about APDIGI14 PutChar behavior, see `ptReadWriteWait` in the APDIGI14 Declarations in §4.E.

APFOSSIL works as described above if `ptPutCharWait` is not set. If it is set, as it is by default, PutChar waits until the character can be added to the output buffer. See `ptPutCharWait` in the APFOSSIL Declarations in §4.D for more details.

APINT14 waits until the character is transmitted or a timeout occurs. If a timeout occurs, you probably have very little visibility into what caused the transmit to fail. For instance, the BIOS functions usually wait until the transmitter is ready and they may also insist that the remote device supply modem signals (DSR and/or CTS). If the transmit fails, you won't know why (nor can you do anything about the DSR/CTS requirements).

Example

```
if OutBuffFree(P) > Length(St) then
  for I := 1 to Length(St) do
    PutChar(P, St[I]);
```

This example assumes use of a buffered device layer (e.g., APUART). Since there is sufficient space in the output buffer, it is safe to make repeated calls to PutChar without checking for errors.

```
I := 1;
repeat
  PutChar(P, St[I]);
  if AsyncStatus = ecOk then
    Inc(I);
until I > Length(St);
```

This example assumes that use of an unbuffered device layer (e.g., APINT14). In this case, it must check AsyncStatus after every call.

See Also

GetChar

OutBuffFree (APCOM)

RestorePort / SavePort

Syntax

```
procedure RestorePort(P : PortRecPtr; var PSR);
```

Purpose

Restore the state of the port from PSR.

Description

If the current device layer allows it, this procedure restores the state of the specified port based on the data stored in PSR, previously initialized by SavePort. Typically, PSR is a variable of type PortSaveRec, but future device layers may require alternate data structures to support this function, hence the untyped parameter.

APINT14, APFOSSIL, and APDIGI14 ignore this call.

Example

See the example for SavePort.

See Also

ActivatePort

DeactivatePort

Syntax

```
procedure SavePort(P : PortRecPtr; var PSR);
```

Purpose

Saves the state of the port in PSR.

Description

If the current device layer allows it, this procedure saves the current state of the specified port in the data structure passed as the PSR parameter. Typically, PSR is a variable of type PortSaveRec, but future device layers may require alternate data structures to support this function, hence the untyped parameter.

APINT14, APFOSSIL, and APDIGI14 ignore this call.

Example

```
var
  P : PortRecPtr;
  PSR : PortSaveRec;
begin
  {initialize P}
  SavePort(P, PSR);
  ...
  RestorePort(P, PSR);
end.
```

Saves the state of the specified port and later restores it.

See Also

ActivatePort

DeactivatePort

Syntax

```
procedure SendBreak(P : PortRecPtr);
```

Purpose

Send a serial line break.

Description

If the current device layer supports it, this procedure sends a break signal to the remote.

A break is not a normal character (it's not really a character at all). A break condition is when the data line is held in a "space" condition for at least one character-time (the amount of time it takes to transmit one character). SendBreak holds the space condition for just over one-character time. A break signal usually has a special meaning for the remote (such as Attention or Interrupt).

APINT14 and APFOSSIL cannot send a break and generate an `ecNotSupported` error.

Example

```
SendBreak(P);
```

Sends a break signal from port P.

See Also

SendLongBreak (APUART)

Syntax

```
procedure SetLine(P : PortRecPtr; Baud : LongInt; Parity : ParityType;  
                  DataBits : DataBitType; StopBits : StopBitType);
```

Purpose

Set the port and the port record with the new values.

Description

This procedure does all that is required at the device level to initialize the serial line for the specified port. When a port is opened, `InitPort` calls `SetLine` for you. The only reason you would call this procedure again is to change the initial line speed and other options.

Baud is the baud rate for this port (50 through 115200). Parity is the parity setting for this port (NoParity through SpaceParity). DataBits is the number of data bits in a byte (5 through 8). StopBits is the number of stop bits after each byte (1 or 2).

If the Baud parameter is 0, then the baud rate is left unchanged. This gives you a means to avoid changing the baud rate, which turns off the port for a relatively lengthy period, when all you need to do is change parity and data bits (e.g., changing from NoParity and 8 data bits to EvenParity and 7 data bits).

In APUART, this function completely clears the UART, sets all new values, and (re)activates the interrupt service routine.

APFOSSIL has a maximum baud rate of 19200 baud.

In APINT14, this function calls the BIOS services to set the line speed and line options.

APINT14 has a maximum baud rate of 9600 baud.

Example

```
InitPort(P, Com1, 1200, NoParity, 8, 1, 500, 500, DefPortOptions);
```

```
...
```

```
SetLine(P, 9600, NoParity, 8, 1);
```

A port is opened and used at 1200 baud. Later the speed is increased to 9600 baud.

SetModem

Syntax

```
procedure SetModem(P : PortRecPtr; DTR, RTS : Boolean);
```

Purpose

Activate DTR and/or RTS signals.

Description

This procedure alters the state of port P's modem signals. These two signals are output signals that the remote can use to judge whether this com port is ready to communicate. The DTR (Data Terminal Ready) signal generally means that this com port is ready and working. The RTS (Request To Send) signal generally means that this com port is ready to receive data; this is the normal hardware flow control signal.

APFOSSIL can set only the DTR signal (it controls RTS internally).

APINT14 cannot set either signal and generates an `ecNotSupported` error.

Example

```
InitPort(P, Com1, 19200, NoParity, 8, 1, 1000, 1000,  
  DefPortOptions and not ptRaiseModemOnOpen);  
{...check for errors...}  
SetModem(P, True, False);
```

Only the DTR signal should be raised, so the `ptRaiseModemOnOpen` option is turned off and the desired signal is set manually.

See Also

GetModem

InitPort

Syntax

```
procedure SetUart(ComName : ComNameType; NewBase: Word;
                  NewIrq, NewVector : Byte);
```

Purpose

Change the standard base address, IRQ, and vector of the specified port.

Description

This procedure allows you to change the base address, IRQ, and vector of the specified port. This is currently only supported for APUART.

From the Async Professional perspective, a com port is defined by its base address and by the IRQ line and interrupt vector it uses. The default settings for all ports are stored in the following arrays: DefBaseAddr, DefIrqMask, and DefComVector. This procedure provides a convenient way to alter these values to account for non-standard UART placement.

ComName is the name of the com port to modify (Com1 through Com36). NewBase is the new base I/O address. NewIrq is the new IRQ mask. NewVector is the new interrupt vector. Specifying 0 for any of the last three parameters means to leave the existing setting alone.

If you specify a NewIrq, you *must* specify a NewVector. Following are the standard vectors for each IRQ:

IRQ	Vector	IRQ	Vector
0	\$8	8	\$70
1	\$9	9	\$71
2	\$A	10	\$72
3	\$B	11	\$73
4	\$C	12	\$74
5	\$D	13	\$75
6	\$E	14	\$76
7	\$F	15	\$77

Once you change the defaults for a particular port, those changes remain in effect for the duration of the program. If you need to change them back, you should save the original values and restore them with another call to SetUart.

This is the only core procedure that does not take a PortRecPtr as a parameter and it is the only core procedure that *must* be called before InitPort. Once a port is open, you cannot change these settings.

When changing the definition of Com3 or Com4, SetUart turns off InitPort's standard "PS/2 handling" by setting PS2DetectMode to PS2Ignore. This prevents InitPort from trying to figure out if it should use the de facto standard or the IBM standard for Com3 and Com4. If you later open a port that needs automatic PS/2 detection, you must set PS2DetectMode back to PS2Auto.

APFOSSIL, APDIGI14, and APINT14 don't support non-standard port addresses or IRQs and generate an ecNotSupported error.

Example

```
SetUart(Com1, $2F0, 0, 0);
InitPort(P, Com4, 2400, NoParity, 8, 1, DefPortOptions);
```

Shows how to account for a Com1 UART at port address \$2F0.

See Also

InitPort

StartTransmitter / SWFlowCtl

Syntax

```
procedure StartTransmitter(P : PortRecPtr);
```

Purpose

Start the transmitter by sending out one character.

Description

This procedure, intended primarily for internal use, is used to restart the transmission process in cases where PutChar is not called. Typically, it is called directly only in cases where you are manually performing the functions performed by PutBlockDirect or SWFlowResume. It's documented mainly for use in writing your own device layer units.

APFOSSIL, APDIGI14, and APINT14 do not require this procedure and simply ignore it if called.

Example

```
if P^.OutBuffCount > 0 then  
  StartTransmitter(P);
```

Force transmission to resume if there's anything to send.

See Also

PutBlockDirect (APCOM)
AbstractPort.PutBlockDirect

SWFlowResume (APCOM)
AbstractPort.SWFlowResume

Syntax

```
{ $IFDEF UseSWFlow }  
procedure SWFlowCtl(P : PortRecPtr; OnChar, OffChar : Char; Resume : Boolean);
```

Purpose

Set the on/off characters or resume a blocked transmitter.

Description

This routine serves two distinct purposes: 1) it sets the characters used for on/off flow control; and 2) it resumes transmitting even if transmits are currently blocked by the remote.

This routine is intended primarily for internal use. You should use the SWFlowSetChars and SWFlowResume routines in APCOM/OOCOM.

APINT14 and APFOSSIL don't support either function and return ecNotSupported.

See Also

SWFlowResume (APCOM)
AbstractPort.SWFlowResume

SWFlowSetChars (APCOM)
AbstractPort.SWFlowSetChars

Syntax

```
{ $IFDEF UseSWFlow }  
function SWFlowGet(P : PortRecPtr) : FlowState;
```

Purpose

Return the state of software flow control.

Description

This function returns one of the following states:

- fsOff - flow control is off
- fsClear - flow control is on but clear
- fsTransWait - the transmitter is blocked
- fsRecWait - the receiver is blocked
- fsAllWait - both transmitter and receive are blocked

This routine is intended primarily for internal use. You should use the SWFlowState function in APCOM/OOCOM.

APINT14 always returns fsOff and the error ecNotSupported.

APFOSSIL and APDIGI14 return only fsOff or fsClear. They can report whether flow control is on, but no details beyond that.

See Also

SWFlowState (APCOM)

AbstractPort.SWFlowState

Syntax

```
{ $IFDEF UseSWFlow }  
procedure SWFlowSet(P : PortRecPtr; Enable : Boolean;  
  BufferFull, BufferResume : Word; Options : Word);
```

Purpose

Start or stop software flow control.

Description

This routine starts or stops software flow control for the current device layer. It is intended primarily for internal use. You should use the SWFlowEnable/SWFlowDisable routines in APCOM/OOCOM.

Options should always be zero because there aren't currently any options for software flow control.

APUART, APFOSSIL and APDIGI14 all provide complete software flow control.

APINT14 doesn't support software flow control and returns ecNotSupported.

See Also

SWFlowDisable (APCOM)
AbstractPort.SWFlowDisable
SWFlowEnable (APCOM)

AbstractPort.SWFlowEnable
SWFlowGet

TransReady / UpdateLineStatus / UpdateModemStatus

Syntax

```
function TransReady(P : PortRecPtr) : Boolean;
```

Purpose

Return True if the transmitter is ready for another character.

Description

This function returns True if the transmitter is ready to accept another character for output.

For APUART, APFOSSIL, and APDIGI14, the transmitter is "ready" if there is room in the output buffer for at least one more character.

For APINT14, the transmitter is "ready" if the UART's Transmitter Holding Register (THR) is empty.

Example

```
if TransReady(P) then  
  PutChar(P, C);
```

If the transmitter is ready, send the character in C.

See Also

PutChar

Syntax

```
function UpdateLineStatus(P : PortRecPtr) : Byte;
```

Purpose

Return the line status register value.

Description

This function returns the line status register value as described in "Registers" in §2.B. It is intended primarily for internal use by the interface layer to assure that it always has an up-to-date copy of the line status register.

APUART, APINT14, and APDIGI14 return all line status bits.

Most FOSSIL drivers return only bit 0 (data ready) and bit 1 (overrun). Due to this shortcoming, the following interface layer routines are not accurate for the APFOSSIL device layer:

CheckLineError, CheckLineBreak, CheckTE, CheckTHRE, and CheckFIFOError. If you are confident that your FOSSIL driver reliably returns other line status bits, you can adjust ValidLineStatus (see §4.D) to account for those bits.

Syntax

```
function UpdateModemStatus(P : PortRecPtr) : Byte;
```

Purpose

Return the modem status register value.

Description

This function returns the modem status register value as described in "Registers" in §2.B. It is intended primarily for internal use by the interface layer to assure that it always has an up-to-date copy of the modem status register.

APUART, APINT14, and APDIGI14 return all modem status bits.

Most FOSSIL drivers return only bit 7 (Data Carrier Detect). Due to this shortcoming, the following interface layer routines are not accurate for the APFOSSIL device layer: CheckCTS, CheckDSR, CheckRI, CheckDeltaCTS, CheckDeltaDSR, CheckDeltaRI, and CheckDeltaDCD. If you are confident that your FOSSIL driver reliably returns other line status bits, you can adjust ValidModemStatus (see §4.D) to account for those bits.

H. Miscellaneous Routines: APMISC

The miscellaneous routines are a group of routines required by both the communication routines and the dearchiving routines. This will typically be the first Async Professional unit in the uses list of your application.

Many of the routines in APMISC, though interfaced, are intended for internal use by various other parts of Async Professional. A few routines that might prove useful in your applications are documented here.

APMISC interfaces the global status variables, AsyncStatus and ArchiveStatus, and all of the error/status constants. It also defines all of the standard status strings associated with those error constants. And, it defines all of the various constants required for Object Professional based stream support.

If you are using Async Professional in conjunction with either Object Professional or Turbo Professional, be sure to define UseOPro or UseTPro in APDEFINE.INC. This will eliminate up to 3000 bytes of duplicate code from APMISC.

Note that if you don't define UseOPro or UseTPro, an internal define is created called Standalone (meaning that Async Professional is being compiled in standalone mode). So, when you see `{IFDEF Standalone}` in any Async Professional documentation, it means that the associated material is included only when you are *not* using Turbo Professional or Object Professional.

Declarations

Constants

```
DefSig : Array[0..5] of Char = 'APF10'#26;
```

Default fax signature.

```
FaxFileExt : String[3] = 'APF';
```

Default fax file extension.

```
{IFDEF UseStreams}
otUartPort           = 400;
otInt14Port          = 401;
otFossilPort         = 402;
otDigi14Port         = 403;
otAbstractProtocol   = 410;
otXmodemProtocol     = 411;
otYmodemProtocol     = 412;
otZmodemProtocol     = 413;
otKermitProtocol     = 414;
otAsciiProtocol      = 415;
otBPlusProtocol      = 416;
otAbstractModem      = 420;
otHayesModem         = 421;
otHstModem           = 422;
otMicrocomModem      = 423;
otNullModem          = 424;
otTerminalEmulator   = 440;
otAnsiEmulator       = 441;
otSmartVirtScreen    = 450;
otTerminalWindow     = 451;
otCaptureTerminalWindow = 452;
```

Unique code numbers for all objects that support stream I/O. You typically won't need to refer to these codes directly, since the supplied stream registration routines do. Values up to 999 are reserved by TurboPower. You may assign higher codes to your own objects.

```

{$IFDEF UseStreams}
ptErrorProc      = 400;  {User supplied port error procedure}
ptAbortProc      = 401;  {User supplied port abort procedure}
ptNoErrorProc    = 402;  {Default error procedure for ports}
ptNoAbortProc    = 403;  {Default abort procedure for ports}
ptWaitCharProc   = 404;  {User supplied port waitchar procedure}
ptNoWaitCharProc = 405;  {Default waitchar procedure for ports}
ptUserStatus     = 410;  {User supplied protocol status procedure}
ptNextFile       = 411;  {User supplied next file procedure}
ptLogFile        = 412;  {User supplied log file procedure}
ptAcceptFile     = 414;  {User supplied accept file procedure}
ptNoUserStatus   = 415;  {Default status procedure}
ptNextFileMask   = 416;  {Default next file procedure}
ptNoLogFile      = 417;  {Default log file procedure}
ptNoAcceptFile   = 418;  {Default accept file procedure}
ptPortPtr        = 419;  {AbstractPortPtr code}
ptUserBack       = 420;  {User supplied protocol background procedure}
ptNoUserBack     = 421;  {Default userback procedure for protocols}

```

Pointer codes for use in stream I/O. Just as object types are assigned unique codes in a stream, so are pointers to static code or data. You'll refer to some of these pointer codes as you register your various user hooks. The appropriate codes are mentioned in the documentation for the Load and Store methods of those objects.

Values up to 999 are reserved by TurboPower. You may assign higher codes to your own pointer types.

```

{$IFDEF UseStreams}
veUartPort      = 0;
veInt14Port     = 0;
veFossilPort    = 0;
veDigi14Port    = 0;
veAbstractProtocol = 1;
veXmodemProtocol = 1;
veYmodemProtocol = 1;
veZmodemProtocol = 1;
veKermitProtocol = 0;
veAsciiProtocol = 0;
veBPlusProtocol = 0;
veAbstractModem = 0;
veHayesModem    = 0;
veCourierModem  = 0;
veMicrocomModem = 0;
veNullModem     = 0;
veTerminalEmulator = 0;
veAnsiEmulator  = 0;
veSmartVirtScreen = 0;
veTerminalWindow = 0;
veCaptureTerminalWindow = 0;

```

Version number codes for all objects that support stream I/O. These numbers will be incremented only when an object is modified in such a way as to require a new Load constructor for it. In that case, the more recent version of the object will get a new version code, and the older version of the object will get a special Load method (Load00, for example) to allow it to be read into the new implementation.

Types

```
{IFDEF Standalone}
LH =
  record
    L, H : Word;
  end;
```

Data type used for typecasting long integers; allows easy access to both the high (H) and low (L) words that make up the LongInt.

```
{IFDEF UseOPro}
RootPtr = ^Root;
Root = object
  constructor Init;
  destructor Done; virtual;
end;
```

All objects in Async Professional are derived from this base object. Root isn't really good for anything by itself—you won't ever want to create an instance of Root. This Root object duplicates the Object Professional Root object when you don't have UseOPro defined in APDEFINE.INC. In fact, the only reason Async Professional needs Root at all is to simplify the conditional compilation directives required to use Object Professional-based streams.

ClearFlag / FlagIsSet

Syntax

```
{IFDEF Standalone}  
procedure ClearFlag(var Flags : Word; FlagMask : Word);
```

Purpose

Clear bit(s) in the parameter Flags. The bits to clear are specified in FlagMask.

Description

This macro clears all bits in Flags that are also set in FlagMask. Note that the Flags parameter must be a variable of type Word.

Example

```
ClearFlag(BitFlags, $0002);
```

Clears bit 1 in BitFlags (\$0002 = 0000000000000010 in binary).

See Also

FlagIsSet

SetFlag

Syntax

```
{IFDEF Standalone}  
function FlagIsSet(Flags, FlagMask : Word) : Boolean;
```

Purpose

Return True if the bit specified by FlagMask is set in Flags.

Description

This macro tests to see if any of the bits set in FlagMask are also set in Flags. In most cases, FlagMask should not have more than one bit set. Although FlagIsSet can tell you if one of them is set, or none of them, it cannot tell you if all of them are set.

Examples

```
FlagIsSet(BitFlags, $0002);
```

Correct. This returns True if bit 1 in BitFlags is set (\$0002 = 0000000000000010 in binary).

```
FlagIsSet(BitFlags, $0003);
```

Probably incorrect. This will not tell you if bits 0 and 1 are both set (\$0003 = 0000000000000011 in binary). It only tells you that at least one of them is set.

See Also

ClearFlag

SetFlag

Syntax

```
{IFDEF Standalone}  
procedure FreeMemCheck(var P; Bytes : Word);
```

Purpose

Deallocate heap space.

Description

This procedure works just like FreeMem, but does nothing if P (assumed to be a pointer variable) is nil. Intended to be used in conjunction with GetMemCheck.

Example

See the example for GetMemCheck.

Syntax

```
{IFDEF Standalone}  
function GetMemCheck(var P; Bytes : Word) : Boolean;
```

Purpose

Allocate heap space.

Description

This function tries to allocate the specified number of Bytes of heap space. If successful, it returns True, and P (assumed to be a pointer variable) points to the block of memory that was allocated. If not, it returns False, and P = nil.

Example

```
var P : Pointer;  
...  
if GetMemCheck(P, 1000) then begin  
  ...  
  FreeMemCheck(P, 1000);  
end;  
Allocates 1000 bytes on the heap and releases it.
```

Syntax

```
{IFDEF Standalone}  
procedure SetFlag(var Flags : Word; FlagMask : Word);
```

Purpose

Set bit(s) in the parameter Flags. The bits to set are specified in FlagMask.

Description

This macro sets all bits in Flags that are set in FlagMask. Note that the Flags parameter must be a variable of type Word.

Examples

```
SetFlag(BitFlags, $0002);  
Sets bit 1 in BitFlags ($0002 = 0000000000000010 in binary).  
SetFlag(BitFlags, $0007);  
Sets bits 2, 1, and 0 in BitFlags ($0007 = 0000000000000111 in binary).
```

See Also

ClearFlag

FlagIsSet

UpdateChecksum / UpdateCrc

Syntax

```
function UpdateChecksum(CurByte : Byte; CheckSum : Byte) : Word;
```

Purpose

Return an updated checksum.

Description

This function is used to calculate checksums. It is intended primarily for internal use by Async Professional, but it may be useful if you are trying to implement an unsupported protocol.

Syntax

```
function UpdateCrc(CurByte : Byte; CurCrc : Word) : Word;
```

Purpose

Return an updated 16-bit CRC.

Description

This function is used to calculate a 16-bit CRC. It is intended primarily for internal use by Async Professional, but it may be useful if you are trying to implement an unsupported protocol.

Syntax

```
function UpdateCrc32(CurByte : Byte; CurCrc : LongInt) : LongInt;
```

Purpose

Return an updated 32-bit CRC.

Description

This function is used to calculate a 32-bit CRC. It is intended primarily for internal use by Async Professional, but it may be useful if you are trying to implement an unsupported protocol.

Syntax

```
function UpdateCrcKermit(CurByte : Byte; CurCrc : Word) : Word;
```

Purpose

Return an updated Kermit CRC.

Description

This function is used to calculate a 16-bit Kermit CRC. It is intended primarily for internal use by Async Professional, but it may be useful if you are trying to implement a protocol that requires a Kermit style CRC.

I. Error Handling

Async Professional implements a consistent method of error handling for all of its procedures and objects. The methodology includes a standardized set of numeric error codes and a function, StatusStr, that converts numeric error codes to strings. Both the error codes and the StatusStr function are in the APMISC unit.

If you are familiar with Object Professional's error handling system, you will have no problem with Async Professional's, since they are essentially identical.

Error codes are stored in variables of type Word. The word is composed of an error type, which is a multiple of 10000; and an error number, which is the remainder. Given an error code, you can determine its type and number with the following statements:

```
ErrorType := Code div 10000;  
ErrorNumber := Code mod 10000;
```

or, putting the code back together again:

```
Code := (10000 * ErrorType) + ErrorNumber;
```

APMISC defines four categories of error types. In descending order of severity, they are: fatal, non-fatal, warning, and message. When a fatal error occurs, the affected object/operation must stop what it is doing. Note that it is generally not necessary for the entire application to halt when a fatal error occurs, although in some cases that may be the only recourse. All errors that occur in an object's constructor are obviously fatal. Common examples of fatal errors occur when there isn't enough memory to allocate a needed buffer, or when a required file cannot be opened.

A non-fatal error means that the requested operation did not occur, but that further operations with the same routines/object are possible. Examples of non-fatal errors include requests to send data to a com port when the output buffer is already full, operations that timed out before they could be completed, or an attempt to dearchive an encrypted file in a ZIP archive.

Warnings are typically associated with invalid user input or failed operations performed at the user's request. For example, a failed search operation in a text editor would produce a warning, as would a validation routine for a data entry screen if the user entered an invalid response. Async Professional does not currently use this category.

The message category is used when a string needs to be passed to a user-written routine, and a message code needs to be passed along with it in case the routine wants to supply a different message instead. Async Professional does not currently use this category.

The other part of an error code, the error number, provides more detail about the error. The lowest error numbers, ranging from 2 (file not found) to 162 (general failure), are the same as Turbo Pascal's I/O error codes. Other numbers are organized loosely in blocks of 1000, with each block indicating a different category of error:

0000-0999	DOS, Turbo Pascal, DOS critical errors
2000-2999	Capacity or environment errors
7000-7999	Warnings, user errors
8000-8999	Programmer errors
9000-9999	Messages and status codes

A detailed listing of all error codes is provided in Appendix B. Note that none of the error codes defined in Async Professional conflicts with an error code defined by Object Professional.

As indicated earlier, the StatusStr function converts each of the Async Professional error codes into a descriptive string. Using this function has advantages over interpreting the error codes yourself. First of all, you don't have to interpret the error codes yourself! And secondly, if you need to translate your application to another language, most of the changes you need to make are in APMISC.PA1.

There are four ways that your program can be informed of errors that occur in programs written with Async Professional:

- constructor errors, reported in AsyncStatus or ArchiveStatus
- error codes stored in AsyncStatus or ArchiveStatus
- errors returned by a function such as Archive.GetLastError
- errors reported to a user-written error handling routine

APMISC declares two global variables of type Word, AsyncStatus and ArchiveStatus. When an object's constructor fails, it returns False or a nil pointer, depending on whether a static or dynamic instance of the object is being created. If your program detects failure, it should immediately check the value of AsyncStatus (if you are using one of the objects related to communications) or ArchiveStatus (if you are using the Lzh, UnLzh, Zip, or UnZip objects), where the constructor will have stored a standard error code indicating the reason for the failure. If no error occurred, AsyncStatus/ArchiveStatus contain zero.

Throughout most of the Async Professional library, you can obtain the code for the last error by checking the AsyncStatus variable. The following units are exceptions to that rule:

```

APARCHIV
APZIP
APLZH
    ArchiveStatus - for all errors

OOARCHIV
OOZIP
OOLZH
    ArchiveStatus - for constructor errors
    Archive.GetLastError - for all other errors

```

Finally, the communications routines (OOP and non-OOP) allow you to install a user-written error handler. This error handler is called by any communication routine that detects an error. When you install such a routine, you avoid having to check AsyncStatus after every procedure or function call.

APPORT declares the AsyncErrorProc type, which defines the parameters passed to the user-written routine. A default error procedure, NoErrorProc, is set by the serial port open routines. This routine ignores all errors. You should replace this with an error handler suitable for your application.

See the discussion of the "User Error Procedure" in APPORT in §4.B for more information and for a sample user error procedure.

Declarations

Constants

```

ecOk      = 0;      {Reset value for AsyncStatus}
...
ecUnPacket = 9993;  {Invalid packet type}

```

Error numbers. See Appendix B for a complete list.

```
epFatal    = etFatal * 10000;  
epNonFatal = etNonFatal * 10000;  
epWarning  = etWarning * 10000;  
epMessage  = etMessage * 10000;
```

Error prefixes added to error numbers before reporting them.

```
etFatal    = 0; {fatal errors}  
etNonFatal = 1; {non-fatal I/O errors}  
etWarning  = 2; {warnings, user input errors, etc.}  
etMessage  = 3; {status information, simple messages}
```

Error types used to classify errors.

Variables

ArchiveStatus : Word;

This global variable is used by the archive-related routines to report errors. Note that OOARCHIV, OOLZH, and OOZIP use this variable only in objects' constructors. When using protected mode DLL's, you cannot reference this variable. See GetArchiveStatus for more information.

AsyncStatus : Word;

This global variable is used by the communication routines to report status and errors back to your program. When using protected mode DLL's, you cannot reference this variable. See GetAsyncStatus for more information.

Syntax

```
function GetArchiveStatus : Word;
```

Purpose

Return the value of the global variable ArchiveStatus.

Description

You need to use this function only if you are using a protected mode DLL. When using DLLs, you can no longer reference the global variable ArchiveStatus, since it is private to the DLL. GetArchiveStatus is provided to return the value of ArchiveStatus. See §2.F for more information on using protected mode DLLs.

Syntax

```
function GetAsyncStatus : Word;
```

Purpose

Return the value of the global variable AsyncStatus.

Description

You need to use this function only if you are using a protected mode DLL. When using DLLs, you can no longer reference the global variable AsyncStatus, since it is private to the DLL. GetAsyncStatus is provided to return the value of AsyncStatus. See §2.F for more information on using protected mode DLLs.

SetArchiveStatus / SetAsyncStatus

Syntax

```
procedure SetArchiveStatus(Status : Word);
```

Purpose

Set the value of the global variable ArchiveStatus.

Description

You need to use this function only if you are using a protected mode DLL. When using DLLs, you can no longer reference the global variable ArchiveStatus, since it is private to the DLL. SetArchiveStatus is provided to set the value of ArchiveStatus. See §2.F for more information on using protected mode DLLs.

Syntax

```
procedure SetAsyncStatus(Status : Word);
```

Purpose

Set the value of the global variable AsyncStatus.

Description

You need to use this function only if you are using a protected mode DLL. When using DLLs, you can no longer reference the global variable AsyncStatus, since it is private to the DLL. SetAsyncStatus is provided to return the value of AsyncStatus. See §2.F for more information on using protected mode DLLs.

Syntax

```
function StatusStr(Code : Word) : String;
```

Purpose

Return a one-line message for the specified error code.

Description

Given an Async Professional error, warning, or message code, this function returns a string that indicates the meaning of the code.

Be forewarned that this routine pulls roughly 6K of code into your program.

Example

```
if AsyncStatus <> 0 then begin  
  WriteLn( StatusStr(AsyncStatus));  
  Exit;  
end;
```

Checks AsyncStatus for an error and displays a suitable message if one is found.

5. The Interface Layer

A. Overview

This chapter covers the interface layer routines of Async Professional. The interface layer provides most of the library functions that your communications program will need. As shown in the architecture diagram in Chapter 2, the interface layer has two "flavors": OOCOM provides the OOP calling format, APCOM provides the non-OOP calling format. The routines in APCOM and OOCOM build on the core routines discussed in §4.G.

The interface layer procedures can be grouped into several categories: receiving data, transmitting data, line control/status routines, modem control/status routines, flow control, and miscellaneous. All routines appear in alphabetical order in the accompanying reference section.

Routines that receive data from the serial port have the prefix "Get" (GetChar, GetString, etc.). Blocks of data are requested by length (when you want a specified number of characters returned) or by delimiter (when you want all characters returned until a specified delimiter character is received). Delimiters are specified as sets and you can request, via the `ptIgnoreDelimCase` option, that the delimiters be treated in a case-insensitive fashion.

Routines that transmit data have the prefix "Put" (PutChar, PutBlock, etc.). Generally, there is a corresponding PutXxx routine for every GetXxx routine (GetString and PutString, GetBlockTimeout and PutBlockTimeout, and so on).

APCOM and OOCOM have a complete set of routines for checking various line control and status settings. Line control means those line parameters that your program sets (baud, data bits, stop bits and parity). Line status means those line status conditions that the UART detects and reports (overrun, parity and framing errors; line breaks; received character ready and transmitter ready).

APCOM and OOCOM also have a complete set of routines for checking various modem control and status settings. Modem control means those modem signals that your program sets (DTR/data terminal ready and RTS/request to send). Modem status means those modem status conditions that the UART detects and reports (DSR/data set ready, CTS/clear to send, RI/ring indicator, and DCD/data carrier detect).

The Async Professional automatic flow control routines are provided in this layer. This includes both hardware flow control and software flow control.

Finally, APCOM and OOCOM include a small grab bag of miscellaneous routines such as WaitForString and ProtocolInProgress.

B. OOP vs. Non-OOP Interface

This is the first level of the Async Professional architecture that implements objects. In fact, this is where Async Professional splits into two different groups of source files: OOP and non-OOP. This split warrants a few comments.

The OOP and non-OOP interfaces are supplied by two separate units: OOCOM for the OOP interface, APCOM for the procedural interface. (The naming convention is to use OOxxx for OOP units and APxxx for non-OOP units.)

The OOP and non-OOP routines are very nearly identical. In fact, the reference section includes only one entry that documents both the OOP and non-OOP versions. Most of the time the names and parameters are identical. The only difference is that the non-OOP parameter list includes a PortRecPtr as the first parameter. In the few cases where routines do have different names, the reference section covers only the OOP or non-OOP entry.

An important difference between the OOP and non-OOP interface layers is that the port objects include a virtual method for each core routine that simply calls the device layer procedure directly. These core methods *do not* use the global procedure pointers. The port objects use methods for *all* port access routines whether they are device layer or interface layer routines.

As a result, with the OOP interface you can have multiple device layers active at one time. The non-OOP interface can have only one device layer active at a time because the core procedures are referenced via the global procedure pointers.

Let's put the interface layer together with the previously discussed abstract and device layers and see what we've got. For the non-OOP interface, your uses list would look like this:

```
uses
    ApMisc, ApPort, ApUart, ApCom;
```

This provides you with access to all library routines in the device and interface layers. Here's what it would look like for an object-oriented program:

```
uses
    ApMisc, ApPort, ApUart, OoCom;
```

The only difference is that APCOM was replaced by OOCOM. Since APPORT is still in the uses list, the core procedure pointers are still available to your program, but you won't use them. Instead, you will call the virtual methods associated with the particular port object you instantiated. (Actually, if UseOOP is defined in APDEFINE.INC, the global procedure pointers aren't compiled at all.)

Let's now look at the corresponding sequences of opening a port and retrieving one character from the input buffer:

```
non-OOP:
var
    P : PortRecPtr;
...
InitPortFast(P, Com1, 9600);
GetChar(P, C);
```

```

OOP:
  var
    ComPort : UartPortPtr;
  ...
  New(ComPort, InitFast(Com1, 9600));
  ComPort^.GetChar(C);

```

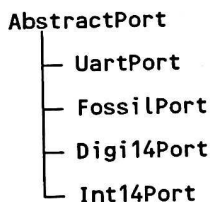
These examples do exactly the same thing (open a port and get one character from the input buffer).

In the non-OOP example, P is a pointer to the standard PortRec structure. InitPortFast initializes the PortRec and sets P to point to that record. Then the core routine GetChar is called to return one character.

In the OOP example, ComPort is a pointer to a UartPort object and is instantiated with a call to the InitFast constructor. Then the GetChar method is called to return one character.

You will probably settle on either the OOP or non-OOP format and thereafter won't have much interest in the format you're not using. Even so, you should study and understand the differences, minor though they are, between the above examples. Doing so will make the rest of this manual, including the many example and demonstration programs, easier to follow.

If you're going to use the OOP format, you'll need to know a bit about the object hierarchy. You can skip the rest of this section if you plan on using the non-OOP format.



The AbstractPort object defines all of the high level routines (like GetCharTimeout and WaitForString). The derived objects simply connect the object to a particular device layer. The UartPort object connects its core virtual methods to APUART's routines; the FossilPort object connects its core virtual methods to APFOSSIL's routines; and so on.

This connection is made through the core virtual methods, those that correspond to the device layer core routines. These methods are listed here, but not documented because they are identical in operation to the core procedures defined in Chapter 4. In those few cases where the core virtual method names are different from the core routine names, the corresponding core routine name is shown in a comment at the right margin. The core methods are:

```

procedure AbstractPort.ActivatePort(Restore : Boolean); virtual;
procedure AbstractPort.BufferFlush(FlushIn, FlushOut: Boolean); virtual;
procedure AbstractPort.BufferStatus(var InFree, OutFree,
                                     InUsed, OutUsed : Word); virtual;

function AbstractPort.CharReady : Boolean; virtual;
procedure AbstractPort.DeactivatePort(Restore : Boolean); virtual;
destructor AbstractPort.Done; virtual;                                {DonePort}
procedure AbstractPort.GetChar(var C : Char); virtual;

```

```

procedure AbstractPort.GetLine(var Baud : LongInt; var Parity : ParityType;
                               var DataBits : DataBitType;
                               var StopBits : StopBitType;
                               FromHardware : Boolean); virtual;

procedure AbstractPort.GetModem(var DTR, RTS : Boolean); virtual;

procedure AbstractPort.GotError(StatusCode : Word); virtual;

{$IFDEF UseHWFlow}
function AbstractPort.HWFlowGet : FlowState; virtual;

{$IFDEF UseHWFlow}
procedure AbstractPort.HWFlowSet(Enable : Boolean;
                                BufferFull, BufferResume : Word;
                                Options : Word); virtual;

procedure AbstractPort.PeekChar(var C : Char; PeekAhead : Word); virtual;

procedure AbstractPort.PutChar(C : Char); virtual;

procedure AbstractPort.RestorePort(var PSR); virtual;

procedure AbstractPort.SavePort(var PSR); virtual;

procedure AbstractPort.SendBreak; virtual;

procedure AbstractPort.SetLine(Baud : LongInt; Parity : ParityType;
                               DataBits : DataBitType;
                               StopBits : StopBitType); virtual;

procedure AbstractPort.SetModem(DTR, RTS : Boolean); virtual;

SetUart--use core procedure

procedure AbstractPort.StartTransmitter; virtual;

{$IFDEF UseSWFlow}
procedure AbstractPort.SWFlowCtl(OnChar, OffChar : Char;
                                Resume : Boolean); virtual;

{$IFDEF UseSWFlow}
function AbstractPort.SWFlowGet : FlowState; virtual;

{$IFDEF UseSWFlow}
procedure AbstractPort.SWFlowSet(Enable : Boolean;
                                BufferFull, BufferResume : Word;
                                Options : Word); virtual;

function AbstractPort.TransReady : Boolean; virtual;

function AbstractPort.UpdateLineStatus : Byte; virtual;

function AbstractPort.UpdateModemStatus : Byte; virtual;

```

Note that there is no method to correspond with the core routine SetUart, since SetUart must be called before a port is opened and that's simply not possible with an object. So in those very few instances where you need the services of SetUart (to deviate from standard port definitions), call the core routine SetUart directly.

Following are the constructors for the UartPort object. The names and parameter lists for FossilPort, Digi14Port, and Int14Port constructors are identical to those for UartPort. Again, the names of the corresponding core procedures appear at the right margin.


```

constructor UartPort.InitCustom(ComName : ComNameType;           {InitPort}
                                Baud : LongInt;
                                Parity : ParityType;
                                DataBits : DataBitType;
                                StopBits : StopBitType;
                                InSize, OutSize : Word;
                                Options : Word);

constructor UartPort.InitKeep(ComName : ComNameType;           {InitPortKeep}
                               InSize, OutSize : Word);

```

Remember, you'll call these methods instead of the core routines. See §4.G, "Core Routines" for information about these methods.

C. The Interface Layer: APCOM/OOCOM

Since APCOM/OOCOM provide the pieces to put together a truly useful program, here is a typical Async Professional program using their capabilities. First the object-oriented version:

```
program ExTerm0;      {EXTERMO.PAS}
uses
  Crt, ApMisc, ApPort, ApUart, OoCom;
var
  UP : UartPort;
  C : Char;
  Finished : Boolean;
begin
  {Open a port}
  if not UP.InitCustom(Com3, 2400, NoParity, 8, 1, 500, 500,
    DefPortOptions) then begin
    WriteLn('Failed to open port: ', AsyncStatus);
    Halt;
  end;

  {Simple terminal--quit on <AltX>}
  Finished := False;
  repeat
    {Process chars to send}
    if KeyPressed then begin
      C := ReadKey;
      if C = #0 then
        Finished := ReadKey = #$2D
      else begin
        while not UP.TransReady do ;
        UP.PutChar(C);
      end;
    end;
  until Finished;
  UP.Done;
end.
```

This example, like many to follow, implements a simple terminal. Given the variety of devices you might be working with (other PCs, modems, instruments, etc.) it's not likely that an example program could be something that everyone can use immediately. A simple terminal illustrates most of the concepts you need to understand, however, so that's used most often as an example. Even so, you'll probably need to modify the call to InitCustom (or InitPort) for your serial port and line parameters.

This example uses a static UartPort object named UP. If the constructor fails, the error code for the failure is in AsyncStatus. If the constructor succeeds, this program drops down to the repeat..until loop. This loop constantly checks for keyboard characters to transmit. Before transmitting, it calls TransReady. Unless you can type faster than the serial port can send out characters, TransReady

should always return True in this example. PutChar is called to transmit the character (or stuff it in the output buffer if the last character hasn't actually been transmitted yet).

After looking for a character to transmit, the program calls CharReady to check for any received characters. If CharReady returns True, the program calls GetChar to retrieve one character from the input buffer and then displays it on the screen. If GetChar returns an AsyncStatus value other than ecOk, then a line error has occurred. If so, the error code is displayed and the input buffer is flushed (since it is not known which character in the input buffer has the error).

Here's the same thing in non-OOP format:

```

program ExTerm;    {EXTERM.PAS}
uses Crt, ApMisc, ApPort, ApUart, ApCom;
var
  P : PortRecPtr;
  C : Char;
  Finished : Boolean;
begin
  {Open a port}
  InitPort(P, Com2, 2400, NoParity, 8, 1, 5000, 5000, DefPortOptions);
  if AsyncStatus <> ecOk then begin
    WriteLn('Failed to open port: ', AsyncStatus);
    Halt;
  end;

  {Simple terminal}
  Finished := False;
  repeat
    {Process chars to send}
    if KeyPressed then begin
      C := ReadKey;
      if C = #0 then
        Finished := ReadKey = #$2D
      else begin
        while not TransReady(P) do ;
        PutChar(P, C);
      end;
    end;
  until Finished;
  DonePort(P);
end.

```

After the call to InitPort, P points to an opened and initialized PortRec. P is passed to all other Async Professional routines. Other than that, this program is functionally equivalent to the OOP version and the discussion of the OOP example applies here as well.

Although these examples are introducing the high level units APCOM and OOCOM, you might have noticed that almost all the functions actually used here are core routines (or core methods). Later examples use more of the high level functions.

Wait vs. No-Wait I/O

Async Professional has two major classes of serial port input/output procedures: wait I/O and no-wait I/O. A wait I/O procedure is one that waits a specified number of clock ticks for an action to occur. A no-wait I/O procedure returns immediately whether or not the requested action could be completed. In Async Professional, any routine that takes a Timeout parameter is a wait I/O procedure; all others are no-wait I/O procedures. Wait I/O procedures can also be distinguished by their names since they all have the format XxxTimeout or WaitForXxx.

GetChar is an example of a no-wait I/O procedure. It always returns immediately whether it actually got a character or not. You would, of course, check AsyncStatus to figure out what happened. Or you could protect the call to GetChar with a call to CharReady:

```
if CharReady(P) then
  GetChar(P, C);
```

This way, a character is guaranteed to be ready when GetChar is called.

GetCharTimeout is an example of a wait I/O procedure. If a character is waiting in the input buffer, GetCharTimeout behaves just like GetChar. However, if there aren't any characters in the input buffer, GetCharTimeout waits up to the specified number of clock ticks for a character to arrive. If a character doesn't arrive within the allotted time, GetCharTimeout returns and reports an error of ecTimeout. ecTimeout is a general error code that means the requested operation did not take place within the specified number of clock ticks.

Here's an example of a wait I/O call:

```
GetCharTimeout(P, C, 182);
case AsyncStatus of
  ecOk      : (got a character);
  ecTimeout : (handle timeout condition);
  else      : (handle line error condition);
end;
```

This procedure waits up to 182 clock ticks (10 seconds) for one character. If a character becomes available before 182 ticks elapse, the procedure returns as soon as it gets the character (i.e., without waiting for the entire 182 ticks).

AsyncStatus is checked to see what happened. If a character arrived successfully within 182 ticks, AsyncStatus returns ecOk. If no character was received, AsyncStatus returns ecTimeout. Any other values mean that a character was received, but some sort of line error occurred.

WaitFor Procedures and the User WaitChar Hook

Some of the wait I/O procedures in Async Professional do more than just return all characters found. Instead, these procedures scan the input stream waiting for specified characters or strings. These are called WaitFor procedures and there are three of them:

```
procedure WaitForChar
procedure WaitForString
procedure WaitForMultiString
```

These procedures don't care about anything in the input stream except what they were asked to look for. A typical use is when a program needs to sample the input stream looking for a prompt (say 'NAME:'). You would handle this with something like:

```
WaitForString(P, 'NAME:', 182);
```

This waits 182 clock ticks for the string 'NAME:' in the input stream. If found, it returns ecOk; otherwise, it returns ecTimeout.

This is fairly straightforward, but it raises the question of what WaitForString should do with the characters that *don't* match 'NAME:'. Should it discard them? Should it display them? Or should it stick them in a buffer somewhere?

Rather than try to force a solution, Async Professional provides a hook called the WaitCharProc. This is a user-defined procedure that all of the WaitFor procedures call for *every* character they receive. This way, you can decide what to do in this situation.

By default, the procedure NoCharWait is assigned as the WaitCharProc. This procedure does nothing. Hence, if the desired behavior is to discard this data, you don't need to take any special action. If you want to do something besides discard the data, such as display it to the screen or capture it in a file, then you must set up your own WaitCharProc.

The WaitCharProc is defined somewhat differently for the OOP and non-OOP interfaces. Here are the definitions:

```
OOP:
  {Procedure for handling chars during WaitForChar/String}
  WaitCharProc = procedure(APPtr : AbstractPortPtr; C : Char);

non-OOP:
  {Procedure for handling chars during WaitForChar/String}
  WaitCharProc = procedure(P : PortRecPtr; C : Char);
```

As you can see, the only difference is that the OOP format has an AbstractPortPtr passed in, whereas the non-OOP format receives a PortRecPtr. Here's an example of a WaitCharProc (OOP variety):

```
{ $F+ }
procedure ShowWaitChars(APPtr : AbstractPortPtr; C : Char);
  { -Displays all received chars during a wait... }
begin
  WriteCharAnsi(C);
end;
...
UP.SetWaitCharProc(ShowWaitChars);
```

This particular example just displays all received data to the screen using the APANSI routine WriteCharAnsi, which handles ANSI escape sequences.

SetWaitCharProc is used to install your WaitChar procedure. Each port object can have its own WaitChar procedure or all port objects can use the same one. Note that this is *not* true for the non-OOP format where there can be only one WaitChar procedure at a time and all ports must use the same one.

A valid WaitChar procedure must be declared FAR using { \$F+ } or the far keyword, it must not be nested inside any other procedure, and it must take exactly the parameters shown in the examples above.

Flow Control

Flow control refers to the ability of either terminal point of a communications link to control the rate of data it's receiving. Flow control is required when different parts of a communication link have different maximum speeds for handling data.

Consider the example of a PC that's receiving data, at a very high speed, from an attached device (say some data collection equipment). Assume this data is arriving so fast that the PC doesn't have time to process and store it all. This situation, continuing unchecked, would eventually overflow the PC's input buffer and data would be lost.

The solution is for the PC to tell the other end of the link, whatever it is, to temporarily stop sending data. Once the PC is caught up, it tells the other end of the link to resume sending data again. In lengthy transactions this stop/start process might be repeated many times. This process is called flow control.

You can look at flow control from two perspectives: receive flow control and transmit flow control. Receive flow control is the ability to tell the other side of a communication link to stop sending data to *you*. Transmit flow control is the ability to honor a request from the other end of a communication link to stop sending data to *it*. In order for your program to fully implement flow control, it must support both of these perspectives.

Flow control comes in two varieties: hardware flow control and software flow control. Hardware flow control relies on signal lines within the serial cable to stop and start the flow. Software flow control relies on special characters in the data stream.

Note that flow control requires cooperation between the device layer and the interface layer. That is, for flow control to work, it *must* be supported at the device layer. Of the provided device layers, only APINT14 does not support flow control.

All discussion of flow control using APCOM/OOCOM centers on what happens between your PC and whatever is directly attached to your PC's serial port. This is rather straightforward when you are attached to an instrument or another computer. However, when you are connected to a modem, which is then connected via a phone link to a remote modem and PC, other issues arise. Notably, when is the flow controlled between the local PC and the local modem and when is the flow controlled between the local PC and the remote PC? We won't attempt to address these issues here. Instead they will be covered in "Modem Flow Control" in §6.C.

Hardware Flow Control

Hardware flow control (sometimes called hardware handshaking) is implemented using control signal lines within the serial cable. The name and meaning of these signals comes from the RS-232 specification (see the Glossary in Appendix A). Since the RS-232 specification describes a connection between a terminal and a modem, these hardware flow control signals are properly called modem control signals and modem status signals.

Many other serial devices—printers, plotters, lab instruments, and so on—also support these "modem" signals for hardware flow control. Unfortunately, many manufacturers that claim to support the RS-232 standard will treat these signals somewhat differently.

Nevertheless, much common ground does exist, particularly among modems. The type of automatic hardware flow control used by Async Professional should work with any popular modem on the market. However, when connecting to instruments, lab equipment, printers, or other computers, you might find slight variations in their interpretation of hardware flow control. Flow control options are provided to help you cope with these situations.

The Async Professional hardware flow control is completely automatic. Once you turn it on, the active device layer manages the modem control output signals for receive flow control and honors the modem status input signals for transmit flow control.

While you don't need to know the gory details about hardware flow control, you do at least need to know the names of the signals to use for flow control and what buffer levels to use to stop and

resume receiving. With that information available, it's easy to turn on hardware flow control. Here's a brief example (OOP variety):

```
New(UP, InitCustom(Com1, 19200, NoParity, 8, 1, BufferSize, BufferSize,
                  DefPortOptions);
if UP = nil then
  {handle error};

UP^.HWFlowEnable(Round(BufferSize * 0.90), Round(BufferSize * 0.10),
                 hfRequireCTS or hfUserRTS);
if AsyncStatus <> ecOk then
  {handle error};
```

HWFlowEnable turns on hardware flow control after the port is opened. This example passes "hfRequireCTS or hfUserRTS" as the flow control options. These options specify standard flow control and will suffice for most of your applications. They mean that Async Professional will require the CTS input signal during transmits (i.e., it won't transmit unless whatever is attached to the serial port has turned on the CTS input signal). Async Professional also turns off the RTS output signal whenever its input buffer exceeds 90% capacity. It turns it on again once its input buffer is drained below 10% capacity.

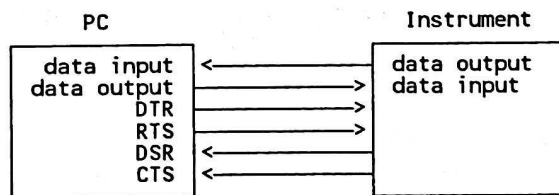
Flow control happens automatically. The application program can continue to issue PutXxx and GetXxx calls without regard for flow control. The only issue you might need to be concerned about is the timeout value used in various PutXxxTimeout routines. That is, you need to deal with the possibility that characters won't be transmitted because they are blocked by flow control. The timeout value needs to be sized such that it accounts for the longest period of time the remote device might block transmission.

In case you need something other than standard flow control, here's the complete list of hardware flow control options (see HWFlowEnable for a more detailed explanation of their meanings).

```
{Hardware flow control options}
hfUseDTR      = $01; {Use DTR for receive flow control}
hfUserRTS     = $02; {Use RTS for receive flow control}
hfRequireDSR  = $04; {Require DSR before transmitting}
hfRequireCTS  = $08; {Require CTS before transmitting}
hfDTRActiveLow = $10; {Make DTR active low}
hfRTSActiveLow = $20; {Make RTS active low}
hfDSRActiveLow = $40; {Make DSR active low}
hfCTSAciveLow = $80; {Make CTS active low}
```

This covers everything you need to know to use hardware flow control with Async Professional. The discussion continues with a more detailed look at hardware flow control. If you're curious, you might want to read on a bit further.

Let's look at the case of a PC that is controlling a laboratory instrument. That is, the PC is sending commands to the instrument and receiving large amounts of data back from it.



The lines between the PC and the instrument represent some of the physical lines within the cable connecting these two devices. The meaning of the lines marked data input and data output should

be obvious (that's where the data flows). The names of the other signals are straight from the RS-232 specification. The arrows in the diagram indicate whether a signal is an output from the PC or an input to the PC. A quick glance shows that DTR and RTS are outputs and DSR and CTS are inputs. Which brings us to the question: what do these things mean and what do they have to do with flow control? The complete names are:

- DTR - data terminal ready
- RTS - request to send
- DSR - data set ready ("data set" is another term for modem)
- CTS - clear to send

These signals have two states: on and off. (You might also see these two states referred to as high and low, raised and lowered, or asserted and de-asserted.)

DTR is commonly turned on by your application to indicate that your program is up and running; but not necessarily ready to receive data. RTS is turned on by your program when it is ready to receive data.

DSR is an input signal from the attached device that, when on, means it is correctly attached and is turned on, but not necessarily ready to receive data. CTS is an input signal from the attached device that, when on, means it is ready to receive data.

The signal pair DTR and DSR aren't usually used in flow control (although Async Professional allows it). Instead, DTR is usually turned on when you open a port just to notify the attached device that the port is now open. Likewise, the attached device usually turns on DSR as soon as you power it on.

The RTS and CTS signals are the ones commonly used to provide hardware flow control. These signals are set and monitored automatically by Async Professional as described earlier. That is, Async Professional lowers RTS when its input buffer fills to the threshold you specified. RTS is raised again when your program has emptied the input buffer below the resume threshold. While transmitting data, Async Professional honors the setting of the CTS input signal—it won't transmit unless the signal is high. Whenever the CTS signal switches from low to high, Async Professional resumes transmitting any data that might be waiting in its output buffer.

Software Flow Control

Software flow control (sometimes called Xon/Xoff flow control) is implemented by assigning special meaning to two characters—a "stop" character and a "start" character. The most commonly used characters are Xoff (ASCII 19) and Xon (ASCII 17). When Async Professional receives an Xoff character, it stops sending data to the remote. When Async Professional receives an Xon character, it resumes sending any data that might be waiting in its output buffer.

Conversely, if the input buffer becomes dangerously full (rises above the specified threshold), Async Professional sends an Xoff to stop the remote from sending data. Once the input buffer is drained sufficiently, Async Professional sends Xon to request the remote to start transmitting again.

As with hardware flow control, all of this is handled automatically by the Async Professional kernel. Here's some example code that turns on software flow control (OOP variety):

```
New(UP, InitCustom(Com1, 19200, NoParity, 8, 1, BufferSize, BufferSize,
                  DefPortOptions));
if UP = nil then {handle error};

UP^.SWFlowEnable(Round(BufferSize * 0.75), Round(BufferSize * 0.25));
if AsyncStatus <> ecOk then {handle error};
```


You should set the buffer full level with more of a safety margin than is necessary for hardware flow control. That's because the remote device may continue to send characters for a brief period after the Xoff is sent. This is due to link propagation delays and the response time of the remote device (how long it takes to recognize the Xoff and stop transmitting).

The same issue applies to the buffer resume level. To keep things going as fast as possible, you shouldn't ever completely empty the input buffer before sending the Xon. Since it might take the remote device a while to respond to the Xon, it should be sent before the input buffer is empty.

This example shows values of 75% of buffer size for the cutoff point and 25% of buffer size for the resume point. These are probably very conservative levels. Don't be too conservative by setting the cutoff and resume points too close together. That would result in "flow control thrashing" where the transmission of many flow control characters would reduce effective throughput.

One-Way Software Flow Control

Async Professional also provides support for one-way software flow control. The two types of one-way flow control are transmit flow control and receive flow control. With transmit flow control, Async Professional's transmitter stops transmitting when it receives an Xoff and resumes when it receives an Xon. With receive flow control, Async Professional's receiver sends an Xoff when the input buffer hits the buffer full level and sends an Xon when the input buffer drops below the buffer resume level.

The SWFlowEnable procedure provides two-way (both transmit and receive) flow control as described above. One-way flow control is enabled via the SWFlowEnableOpt procedure which is identical to SWFlowEnable except that it takes an additional parameter called Opt.

To enable one-way software flow control, call SWFlowEnableOpt and pass sfTransmitFlow or sfReceiveFlow (see §4.B for a description of these two options). Passing both, (sfTransmitFlow + sfReceiveFlow), is acceptable and has the same effect as calling SWFlowEnable: two-way software flow control is enabled.

One-way software flow control is supported by all devices layers that support software flow control: APUART, APFOSSIL and APDIGI14. One-way flow control is not supported by APINT14.

See COMTEST/COMTESTO for examples of using one-way and two-way software flow control.

Status Buffering

Status buffering provides enhanced detection of line errors. Without status buffering, line errors are rather ambiguous. For example, when GetString (or any GetXxx routine) returns a line error, you don't have any way of knowing which character is bad. It could be any character in the returned string, or it could even be a character still in the input buffer. Since you don't know, all of the input data is suspect.

In many cases this doesn't cause any great hardship. Protocols, for one, rely on a block check character to verify data integrity. If the block check character is bad, the whole block is discarded and must be sent again. However, other applications (like some real-time data acquisition systems) might not have the luxury of managing data in blocks. Such applications require that each received character be tagged as good or bad. That's what status buffering does.

When you turn on status buffering, via EnableStatusBuffer, a buffer is created to hold a status byte for every character in the input buffer. (Note that any bytes already waiting in the input buffer will have their status bytes set to zero.) As new characters are received at the serial port, the kernel saves the status of each character in the status buffer. The possible values are:

- ecOk - character received without errors
- ecFraming - character had a framing error
- ecOverrun - character had an overrun error
- ecParity - character had a parity error

Once status buffering is enabled, you continue to check for errors (via AsyncStatus) after all GetXxx calls, just as though status buffering were off. The difference is that when you detect an error, it is guaranteed to apply to the current character. For the routines that return multiple characters (like GetString), the error is guaranteed to apply to the *last* character returned.

Here's an example of turning on and using status buffering (OOP variety):

```
{Open a port}
New(UP, InitFast(Com2, 2400));
{... check for errors ...}

{$IFDEF UseStatusBuffer}
{Turn on status buffering}
UP^.EnableStatusBuffer;
if AsyncStatus <> ecOk then
  Abort('Failed to enable status buffering ', AsyncStatus);
{$ENDIF}

{Simple terminal}
Finished := False;
repeat
  {Process transmitted characters}
  ...

  {Process received characters}
  if UP^.CharReady then begin
    UP^.GetChar(C);
    if AsyncStatus <> ecOk then
      WriteLn('M'J'LineError ', AsyncStatus, ' character discarded')
    else
      Write(C);
  end;
until Finished;
```

Status buffering is turned on with the call to EnableStatusBuffer. After that, all line errors refer to the last character retrieved from the input buffer. The previous examples flushed the input buffer whenever a line error occurred (since it was not possible to determine which character in the input buffer was bad). This example just discards the current character and keeps going.

Since status buffering is somewhat of a specialty feature, it can be turned on and off via the UseStatusBuffer define in APDEFINE.INC. Turning it off saves a few hundred bytes of code and, perhaps more important, saves a few processing cycles within the interrupt service routine.

APCOM Declarations

Types

WaitCharProc = procedure(P : PortRecPtr; C : Char);

Procedure called by WaitForChar, WaitForString, and WaitForMultiString to allow received characters to be displayed or stored.

OOCOM Declarations

Types

```
AbstractPortPtr = ^AbstractPort;  
AbstractPort =  
  object(Root)  
    PR          : PortRecPtr;  {Pointer to port record}  
    ...  
  end;
```

Abstract port object. Defines data and methods to be shared in common by UartPort, FossilPort, Digi14Port, Int14Port, and other (future or user-written) device layer objects. The PR field points to the corresponding PortRec.

```
{ $IFDEF UseUart }  
UartPortPtr = ^UartPort;  
UartPort =  
  object(AbstractPort)  
    ...  
  end;
```

Object derived from AbstractPort to implement an APUART-based device layer.

```
{ $IFDEF UseFossil }  
FossilPortPtr = ^FossilPort;  
FossilPort =  
  object(AbstractPort)  
    ...  
  end;
```

Object derived from AbstractPort to implement a FOSSIL-based device layer.

```
{ $IFDEF UseDigi14 }  
Digi14PortPtr = ^Digi14Port;  
Digi14Port =  
  object(AbstractPort)  
    ...  
  end;
```

Object derived from AbstractPort to implement a DigiBoard-based device layer.

```
{ $IFDEF UseInt14 }  
Int14PortPtr = ^Int14Port;  
Int14Port =  
  object(AbstractPort)  
    ...  
  end;
```

Object derived from AbstractPort to implement an APINT14-based device layer.

```
WaitCharProc = procedure (APPtr : AbstractPortPtr; C : Char);
```

Procedure called by WaitForChar, WaitForString, and WaitForMultiString to allow rejected characters to be displayed or stored.

BlockReady / ChangeBaud

Syntax

```
function BlockReady(P : PortRecPtr; ExpectedLen : Word;  
                  DelimSet : CharSet) : Boolean;  
  
function AbstractPort.BlockReady(ExpectedLen : Word;  
                                DelimSet : CharSet) : Boolean;
```

Purpose

Return True if a block of a specified length or ending with a character in DelimSet is ready.

Description

This function returns True if 1) a block of at least ExpectedLen characters is waiting in the input buffer or 2) a block of characters delimited by one of the characters in DelimSet is in the input buffer. If ExpectedLen > 0 and DelimSet is not empty, then BlockReady will return True if either condition is met.

Example

```
if BlockReady(P, 10, []) then  
  GetBlock(P, Block, 10, ReceivedLen, []);
```

If a block of 10 characters is ready, read it with GetBlock. Since an empty set is passed for DelimSet, no delimiter checking is done. When using objects, the P parameters would be omitted.

See Also

GetBlock

Syntax

```
procedure ChangeBaud(P : PortRecPtr; NewBaud : LongInt);  
procedure AbstractPort.ChangeBaud(NewBaud : LongInt);
```

Purpose

Change the baud rate of the specified port.

Description

This procedure changes the baud rate of the specified port to NewBaud. In general, the valid range for baud rates is 50 to 115200; however, some device layers may further restrict this range (for example, the maximum baud rate for APINT14 is 9600). If the specified baud rate is out of bounds, an `ecInvalidBaudRate` error is generated.

Examples

```
InitPortFast(P, Com4, 2400);  
...  
ChangeBaud(P, 1200);
```

Initialize the port at 2400 baud, later changing to 1200 baud.

```
UP.InitFast(Com4, 2400);  
...  
UP.ChangeBaud(1200);
```

OOP equivalent of the previous example (UP is of type UartPort).

See Also

SetLine (core)

Syntax

```
procedure ChangeBufferSizes(P : PortRecPtr; NewInSize, NewOutSize : Word);  
procedure AbstractPort.ChangeBufferSizes(NewInSize, NewOutSize : Word);
```

Purpose

Change input/output buffer sizes.

Description

This procedure changes the sizes of the input and/or output buffers used by the specified port. NewInSize is the new size for the input buffer and, if status buffering is on, the new size for the status buffer. NewOutSize is the new size for the output buffer. Note that the existing contents of a buffer are destroyed when its size is changed, so ChangeBufferSizes should not be called while a communications session is in progress.

If either size parameter is 0, the existing buffer is left unchanged, and if either one is not in the legal range (10 through 65521), an `ecInvalidArgument` error is generated. Other possible errors are `ecNotBuffered` (the current device layer is not buffered) and `ecOutOfMemory` (not enough memory to allocate the new buffer(s)—the old buffer(s) will remain in place).

Examples

```
InitPort(P, Com1, 1200, NoParity, 8, 1, 1024, 1024, DefPortOptions);  
...  
ChangeBufferSizes(P, 0, 2048);
```

Open a port with 1K input and output buffers. Later change the size of the output buffer to 2K, leaving the input buffer alone.

```
UP.InitCustom(Com1, 1200, NoParity, 8, 1, 1024, 1024, DefPortOptions);  
...  
UP.ChangeBufferSizes(0, 2048);
```

OOP equivalent of the previous example (UP is of type `UartPort`).

See Also

`InitPort` (core)

Syntax

```
procedure ChangeDataBits(P : PortRecPtr; NewDataBits : DataBitType);  
procedure AbstractPort.ChangeDataBits(NewDataBits : DataBitType);
```

Purpose

Change the data bit setting for the specified port.

Description

This routine changes the data bit line setting (byte size) for the specified port to NewDataBits (5 through 8).

Examples

```
InitPortFast(P, Com4, 2400);  
{...use the port...}  
ChangeDataBits(P, 7);
```

Changes P from the default data bit value of 8 to the new value of 7.

```
UP.InitFast(Com4, 2400);  
...  
UP.ChangeDataBits(7);
```

OOP equivalent of the previous example (UP is of type `UartPort`).

See Also

`SetLine` (core)

ChangeParity / ChangeStopBits

Syntax

```
procedure ChangeParity(P : PortRecPtr; NewParity : ParityType);  
procedure AbstractPort.ChangeParity(NewParity : ParityType);
```

Purpose

Changes parity for the specified port.

Description

This routine changes the parity setting for the specified port to NewParity (NoParity through SpaceParity). If the device layer in use does not support the specified parity type, an error of `ecInvalidParity` is generated.

Examples

```
InitPortFast(P, Com4, 2400);  
ChangeParity(P, OddParity);
```

Changes P from the default setting of NoParity to OddParity.

```
UP.InitFast(Com4, 2400);
```

```
...
```

```
UP.ChangeParity(OddParity);
```

OOP equivalent of the previous example (UP is of type `UartPort`).

See Also

`SetLine` (core)

Syntax

```
procedure ChangeStopBits(P : PortRecPtr; NewStopBits : StopBitType);  
procedure AbstractPort.ChangeStopBits(NewStopBits : StopBitType);
```

Purpose

Change the stop bit setting for the specified port.

Description

This routine changes the stop bit setting for the specified port to NewStopBits (1 or 2). Note that, if the data bit size is 5, then a request for 2 stop bits is interpreted as a request for 1.5 stop bits.

Examples

```
InitPortFast(P, Com4, 2400);
```

```
...
```

```
ChangeStopBits(P, 2);
```

Changes P from the default setting of 1 stop bit to use 2 stop bits.

```
UP.InitFast(Com4, 2400);
```

```
...
```

```
UP.ChangeStopBits(2);
```

OOP equivalent of the previous example (UP is of type `UartPort`).

See Also

`SetLine` (core)

Syntax

```
function CheckCTS(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckCTS : Boolean;
```

Purpose

Return True if CTS is high.

Description

This function returns True if port P detects that the remote has set CTS (Clear To Send). CTS is an input signal that, when detected, indicates that data can be transmitted to the remote (this signaling process is sometimes referred to as hardware flow control or hardware handshaking). Even if you're not using automatic hardware flow control, you may want to check this signal before transmitting.

Most Fossil drivers do not yet support this function.

Example

```
if CheckCTS(P) then  
  PutString(P, 'ABCD')  
else  
  WriteLn('Error. Remote not ready');
```

If the remote is ready to receive characters, send the string 'ABCD' to it. When using objects, the P parameters would be omitted.

See Also

CheckDeltaCTS

CheckDSR

Syntax

```
function CheckDataReady(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckDataReady : Boolean;
```

Purpose

Return True if DR (Data Ready) is high.

Description

This function returns the current setting of the Data Ready (DR) bit of the line status register of a UART. The data ready bit indicates that a character is in the received character register and should be read as soon as possible.

Under normal circumstances you would never need to reference this bit directly (you can use CharReady instead). In fact, when using a buffered device layer such as APUART, this bit is always clear. That is, the kernel always sees and removes the character (clearing the Data Ready bit) before you have a chance to see it.

Even when using an unbuffered device layer such as APINT14, the reliability of this bit is questionable. This function is provided in case it is needed when using future built-in or user-written device layers.

Example

```
if CheckDataReady(P) then  
  GetChar(P, Ch);
```

If the Data Ready signal is high, read the waiting character. When using objects, the P parameters would be omitted.

See Also

CharReady (core)

CheckDCD / CheckDeltaCTS

Syntax

```
function CheckDCD(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckDCD : Boolean;
```

Purpose

Return True if DCD is high.

Description

This function returns True if port P detects that the remote has set DCD (Data Carrier Detect). DCD is usually set only if the remote device is a modem. Your modem raises this signal to tell you that it has a connection with another modem. APMODEM provides some services to make this process a little more convenient (see §6.C).

While the two modems are connected, the DCD signal remains high. If either modem hangs up, your modem lowers this signal. So, when your remote connection involves a modem, you may want to check this signal periodically to make sure that the modems are still connected. Like so:

```
{...establish connection via modem...}  
while CheckDCD(P) do begin  
  while CharReady(P) do  
    {...process received data...}  
  while TransReady(P) do  
    {...transmit data...}  
end;
```

This loop processes the link for as long as DCD is high and therefore handles the case where the remote modem hangs up unexpectedly.

Example

```
if not CheckDCD(P) then begin  
  WriteLn('Error. Remote modem hung up');  
  Exit;  
end;
```

Abort if DCD is not set. When using objects, the P parameter would be omitted.

Syntax

```
function CheckDeltaCTS(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckDeltaCTS : Boolean;
```

Purpose

Return True if DeltaCTS is high.

Description

This function returns True if the DeltaCTS bit is set in the modem status register. The DeltaCTS bit indicates that the Clear To Send signal changed since the last time the register was read.

If it is important for you to know if the CTS changed (rather than what its current value is), then you should call this function instead of CheckCTS. CheckCTS only tells you the current status of CTS. However, it could have been dropped and then raised again (by the remote), and CheckCTS can't tell you that. CheckDeltaCTS can. This might be useful if you want to know if the remote device is using hardware handshaking to slow down the local port.

Most Fossil drivers do not yet support this function.

Example

```
if CheckDeltaCTS(P) then  
  {...CTS changed at least once: do something...}
```

Do something if CTS has changed. When using objects, the P parameter would be omitted.

Syntax

```
function CheckDeltaDCD(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckDeltaDCD : Boolean;
```

Purpose

Return True if DeltaDCD is high.

Description

This function returns True if the DeltaDCD bit is set in the modem status register. The DeltaDCD bit indicates that the Data Carrier Detect signal changed since the last time the register was read.

If it is important for you to know if the DCD changed (rather than what its current value is), then you should call this function instead of CheckDCD. CheckDCD only tells you the current status of DCD. However, it could have been dropped and then raised again (by the remote), and CheckDCD can't tell you that. CheckDeltaDCD can.

Most Fossil drivers do not yet support this function.

Example

```
if CheckDeltaDCD(P) then {...do something...}  
Do something if DCD has changed.
```

See Also

CheckDCD

Syntax

```
function CheckDeltaDSR(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckDeltaDSR : Boolean;
```

Purpose

Return True if DeltaDSR is high.

Description

This function returns True if the DeltaDSR bit is set in the modem status register. The DeltaDSR bit indicates that the Data Set Ready signal changed since the last time the register was read.

If it is important for you to know if the DSR changed (rather than what its current value is), then you should call this function instead of CheckDSR. CheckDSR only tells you the current status of DSR. However, it could have been dropped and then raised again (by the remote), and CheckDSR can't tell you that. CheckDeltaDSR can.

Most Fossil drivers do not yet support this function.

Example

```
if CheckDeltaDSR(P) then {...do something...}  
Do something if DSR has changed.
```

See Also

CheckDSR

CheckDeltaRI / CheckDSR

Syntax

```
function CheckDeltaRI(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckDeltaRI : Boolean;
```

Purpose

Return True if DeltaRI is high.

Description

This function returns True if the DeltaRI bit is set in the modem status register. The DeltaRI bit indicates that the Ring Indicator signal changed since the last time the register was read. (The technical name for this signal is Trailing Edge Ring Indicator or TERI. We use the name DeltaRI, for delta ring indicator, since it fits in better with our nomenclature.)

When checking for incoming calls, this is definitely a better choice than CheckRI, since the ring indicator bit toggles on and off as controlled by the modem. For some modems, the "on" time is very short. However, the DeltaRI bit is set on the very first ring and stays set until this register is read. So, a check for DeltaRI always shows you when the phone is ringing (or was ringing).

Most Fossil drivers do not yet support this function.

Example

```
NewTimerSecs(ET, 30);  
repeat  
until CheckDeltaRI(P) or TimerExpired(ET);
```

Waits up to 30 seconds for an incoming phone call. When using objects, the P parameter would be omitted.

See Also

AnswerModem (APMODEM) CheckRI
AbstractModem.AnswerModem

Syntax

```
function CheckDSR(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckDSR : Boolean;
```

Purpose

Return True if DSR is high.

Description

This function returns True if the specified port detects that the remote has set DSR (Data Set Ready). DSR is an input signal that a remote issues to indicate that it is attached and active.

You may want to check this signal sometime before transmitting (and periodically thereafter) to see if the remote is still active.

Most Fossil drivers do not yet support this function.

Example

```
if CheckDSR(P) then  
  PutString(P, 'ABCD')  
else  
  WriteLn('Error. Remote not ready');
```

If the remote is ready to receive characters, send the string 'ABCD' to it. When using objects, the P parameters would be omitted.

See Also

CheckCTS CheckDeltaDSR

Syntax

```
function CheckFifoError(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckFifoError : Boolean;
```

Purpose

Return True if the FIFO error bit is set.

Description

This function returns True if the FIFO error bit is set in the line status register. This bit indicates that a character in the FIFO buffer was received with a line error (overflow, parity error, or framing error). This error won't be revealed to the rest of the UART (and, consequently, to you) until the error character reaches the top of the FIFO.

The most likely use for this procedure is in deciding whether or not to flush the FIFO if you detect a line error while processing received data. You might play it safe and flush everything, or you might decide to check the FIFO first and flush it only if it has line errors as well. In practice, you can probably skip this step safely since the data is never in the FIFO very long.

Most Fossil drivers do not yet support this function.

Examples

```
if CheckLineError(P) then begin  
  {Flush the input buffer}  
  FlushInBuffer(P);  
  if CheckFifoError(P) then begin  
    {Flush the FIFO buffer as well}  
    SetFifoBuffering(P^.BaseAddr, False, 0);  
    SetFifoBuffering(P^.BaseAddr, True, 14);  
  end;  
end;
```

Flushes the FIFO (by turning it off and on) whenever both it and the input buffer have line errors.

```
if UP.CheckLineError then begin  
  UP.FlushInBuffer;  
  if UP.CheckFifoError then begin  
    SetFifoBuffering(UP.PR^.BaseAddr, False, 0);  
    SetFifoBuffering(UP.PR^.BaseAddr, True, 14);  
  end;  
end;
```

OOP equivalent of the previous example (UP is of type UartPort).

See Also

SetFifoBuffering (APUART)

CheckLineBreak / CheckLineError

Syntax

```
function CheckLineBreak(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckLineBreak : Boolean;
```

Purpose

Return True if a break was received (BI bit is high).

Description

This function returns True if a break signal was received by the specified port. Note that CheckLineBreak clears the flag that indicates a break was received.

Most Fossil drivers do not yet support this function.

Example

```
if CheckLineBreak(P) then  
  Exit;
```

Exit if a break signal was received. When using objects, the P parameter would be omitted.

See Also

SendBreak (core)

Syntax

```
function CheckLineError(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckLineError : Boolean;
```

Purpose

Return True if any error bits are set in the line status register.

Description

This function returns True if the specified port has received any line errors. Note that CheckLineError does *not* reset the error indicator. Subsequent calls to CheckLineError will also return True.

Usually, the next step is to call GetLineError, which returns the exact line error. Calling GetLineError clears the error indicator.

Since all GetXxx routines also return line errors, you don't need to make a special call when getting characters. The purpose of CheckLineError/GetLineError is to handle line errors when you aren't making some other specific GetXxx call.

Most Fossil drivers do not yet support this function.

Example

```
if CheckLineError(P) then  
  WriteLn('Got line error: ', GetLineError(P));
```

Check for and report line errors. When using objects, the P parameter would be omitted.

See Also

GetLineError

Syntax

```
function CheckRI(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckRI : Boolean;
```

Purpose

Return True if RI is high.

Description

This function returns True if the Ring Indicator bit is currently set. The ring indicator bit fluctuates as the incoming rings are detected. When trying to detect a ringing phone, it's much better to call CheckDeltaRI.

Most Fossil drivers do not yet support this function.

Example

```
NewTimerSecs(ET, 30);  
repeat  
until CheckRI(P) or TimerExpired(ET);
```

Waits up to 30 seconds for an incoming phone call. When using objects, the P parameter would be omitted.

See Also

CheckDeltaRI

Syntax

```
function CheckTE(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckTE : Boolean;
```

Purpose

Return True if the transmitter is empty.

Description

This function returns True if the UART transmitter is completely empty (the holding register and the shift register are both empty). Use this call to make sure that you aren't leaving the last character "in the pipe" as you close the port.

Most Fossil drivers do not yet support this function.

See Also

CheckTHRE

DonePort (core)

Syntax

```
function CheckTHRE(P : PortRecPtr) : Boolean;  
function AbstractPort.CheckTHRE : Boolean;
```

Purpose

Return True if the transmitter holding register is empty.

Description

This function returns True if the UART's transmitter holding register is empty. When this register is empty, it's OK to send another character to the UART.

Both APUART and APINT14 make this check internally, so there is little reason to use this procedure.

Most Fossil drivers do not yet support this function.

DisableStatusBuffer

Syntax

```
{IFDEF StatusBuffering}
procedure DisableStatusBuffer(P : PortRecPtr);
procedure AbstractPort.DisableStatusBuffer;
```

Purpose

Disable the input status buffer.

Description

This procedure turns off the character-by-character tracking of line errors. With status buffering disabled, calls to GetChar and CheckLineError return ambiguous line errors (overrun, parity, or framing errors). That is, you have no way of knowing whether a particular error, say a framing error, is associated with the character just returned by GetChar or some other character still in the input buffer.

Example

```
DisableStatusBuffer(P);
repeat
  {...process received and transmitted chars...}
  if CheckLineError(P) then
    {...flush the input buffer...};
until False;
```

With status buffering off, the error handling logic must be to flush the entire input buffer (since you don't know which character is bad).

See Also

EnableStatusBuffer

Syntax

```
procedure DrainOutBuffer(P : PortRecPtr; Timeout : Word);  
procedure AbstractPort.DrainOutBuffer(Timeout : Word); virtual;
```

Purpose

Delay until the output buffer is drained or Timeout.

Description

This procedure pauses until the output buffer is empty or Timeout clock ticks have elapsed.

In a buffered environment, you have no control over when characters are actually transmitted. DrainOutBuffer gives you a mechanism to allow some time for the transmission to be completed.

Examples

Non-OOP

```
PutString(P, 'ABCDEFGH');  
DonePort(P);
```

OOP

```
UP.PutString('ABCDEFGH');  
UP.Done;
```

(UP is of type UartPort.) In this example it's very likely that only the first one or two characters of the string will actually be transmitted. PutString simply stuffs the string in the output buffer and returns immediately. The kernel is then responsible for transmitting the characters, one at a time, until the buffer is empty. The call to DonePort will almost certainly be executed long before the kernel has a chance to send all of the characters. A better way to do this is:

Non-OOP

```
PutString(P, 'ABCDEFGH');  
DrainOutBuffer(P, 1092);  
while not CheckTE(P) do ;  
DonePort(P);
```

OOP

```
UP.PutString('ABCDEFGH');  
UP.DrainOutBuffer(1092);  
while not UP.CheckTE do ;  
UP.Done;
```

DrainOutBuffer is given a maximum of 60 seconds ($60 \times 18.2 = 1092$) to insure that the characters in the output buffer were sent. The repeated calls to CheckTE insure that the last byte that was moved from the output buffer to the UART actually gets sent to the remote.

EnableStatusBuffer / FaxInProgress

Syntax

```
{IFDEF StatusBuffering}  
procedure EnableStatusBuffer(P : PortRecPtr);  
procedure AbstractPort.EnableStatusBuffer;
```

Purpose

Enable the input status buffer.

Description

This routine allocates a status buffer and begins character-by-character tracking of line errors (overrun, parity, or framing errors). All characters received after this call will have a status byte attached to them. That status byte tells you whether or not that character had any line errors. The status byte is checked whenever GetChar is called. Errors are returned in AsyncStatus.

EnableStatusBuffer carries with it two penalties. First, an additional buffer (the same size as the input buffer) is allocated on heap. Second, a few extra instructions are executed in the kernel (trivial, but if you're running close to the limit, this could put you over). GetChar and all related routines also execute a few more instructions.

Status buffering is supported by the APUART device layer only.

Example

```
EnableStatusBuffer(P);  
...  
GetChar(P, C);  
case AsyncStatus of  
  ecOk : Write(C);  
  ecBufferIsEmpty : {do nothing};  
  else WriteLn('Error receiving ', C, ' Status = ', AsyncStatus);  
end;
```

Status buffering is on, so each character can be checked for line errors. When using objects, the P parameters would be omitted.

See Also

DisableStatusBuffer

Syntax

```
function FaxInProgress(P : PortRecPtr) : Boolean;  
function AbstractPort.FaxInProgress : Boolean;
```

Purpose

Return True if this port is currently processing a fax.

Description

This function returns True if the specified port is in the process of sending or receiving a fax. It's primarily useful in user-written error handlers for programs that send and receive faxes.

Example

See the examples for SetErrorProc.

See Also

SetErrorProc

Syntax

```
procedure FlushInBuffer(P : PortRecPtr);  
procedure AbstractPort.FlushInBuffer; virtual;
```

Purpose

Flush the input buffer.

Description

This routine marks the input buffer as empty. It does nothing if input isn't buffered.

Example

```
if CheckLineError(P) then begin  
  WriteLn('Got line error, status = ', AsyncStatus);  
  FlushInBuffer(P);  
end;
```

Flushes all data in the input buffer in response to a line error. When using objects, the P parameters would be omitted.

See Also

CheckTE

Syntax

```
procedure FlushOutBuffer(P : PortRecPtr);  
procedure AbstractPort.FlushOutBuffer; virtual;
```

Purpose

Flush the output buffer.

Description

This routine marks the output buffer as empty. It does nothing if output isn't buffered.

You might use this procedure if your remote reports an error to you and you need to back up to a specific point and re-transmit. In that case, there's no purpose in sending data that might still be in the output buffer.

Example

```
if {Got error from remote} then begin  
  FlushOutBuffer(P);  
  {resend data}  
end;
```

Flushes all data in the output buffer in response to an error reported by the remote. When using objects, the P parameter would be omitted.

ForceBufferLimits / GetBaseAddr

Syntax

```
procedure ForceBufferLimits(P : PortRecPtr; NewInLimit, NewOutLimit: Word);  
procedure AbstractPort.ForceBufferLimits(NewInLimit, NewOutLimit: Word);
```

Purpose

Force new buffer limits.

Description

This function changes the limits of the existing input and output buffers. Unlike `ChangeBufferSizes`, `ForceBufferLimits` does not actually deallocate the old buffers or allocate new ones. Instead, it sets a new limit on the amount of the existing buffer used by Async Professional.

A likely purpose for this routine is to temporarily reduce the size of the buffers during terminal emulation (where you don't want to get too far ahead of the remote session) and then increase the buffer limits back to their maximum before starting a protocol.

Generally, you should use `ForceBufferLimits` to make small to moderate buffer size changes (say, changes of 2K to 8K) or for larger changes if you don't mind the temporary wasted buffer space that remains allocated but unused. Use `ChangeBufferSizes` when making larger buffer changes or when heap space is scarce.

Note: `ForceBufferLimits` cannot detect an attempt to set a buffer limit beyond the allocated buffer size. If, for example, you originally allocated a 2048 byte input buffer, `ForceBufferLimits` will accept a request to change this to 4096—causing Async Professional to write beyond the end of the allocated buffer. It's your responsibility to assure that the buffer sizes you pass to `ForceBufferLimits` are within the size of the original buffer allocation.

Example

```
UP.InitFast(Com1, 2400);  
UP.ForceBufferLimits(128, 128);
```

Opens a port with the default buffer sizes (2048 for input, 2078 for output) and immediately reduces the buffer limits to 128 for each.

See Also

`ChangeBufferSizes`

Syntax

```
function GetBaseAddr(P : PortRecPtr) : Word;  
function AbstractPort.GetBaseAddr : Word;
```

Purpose

Return the UART base address of this port.

Description

This function returns the base address of the UART associated with the currently open port. If the port doesn't use a UART (as with `Digi14Port` or `FossilPort`), `GetBaseAddr` returns 0.

APUART contains several routines, like the FIFO management routines, that are specific to UARTs and therefore not part of the core routines. Such routines require a base address parameter instead of a `PortRecPtr` (or port object) and `GetBaseAddr` provides a convenient way of obtaining that base address value.

Example

```
var  
  BaseAddr : Word;  
...  
UP.InitFast(Com1, 2400);  
BaseAddr := UP.GetBaseAddr;  
SetFifoBuffering(BaseAddr, True, 4);
```

Gets the base address of Com1 and turns on FIFO buffering for that UART.

Syntax

```

procedure GetBlock(P : PortRecPtr; var Block; ExpectedLen : Word;
                  var ReceivedLen : Word; DelimSet : CharSet);
procedure AbstractPort.GetBlock(var Block; ExpectedLen : Word;
                              var ReceivedLen : Word;
                              DelimSet : CharSet); virtual;

```

Purpose

Return a block of a specified length or ending with a character in DelimSet.

Description

This procedure attempts to return either 1) a block of ExpectedLen characters or 2) a block of characters delimited by one of the characters in DelimSet. If ExpectedLen > 0 and DelimSet is not empty, GetBlock succeeds if either of these conditions can be met. If ExpectedLen is 0 and DelimSet is empty, an ecInvalidArgument error is generated.

Block is a variable supplied by the caller that must be large enough to hold at least ExpectedLen characters or the delimited block. ReceivedLen is set to the number of characters actually returned, even if errors are encountered.

If the current device layer is buffered, the ptReturnPartialGets option is off, and neither of these conditions can be met, an ecBufferIsEmpty error is generated and no characters are returned in Block. If the ptReturnPartialGets option is on, GetBlock returns as many characters as it can and an ecBlockIncomplete error. If there are no characters to return, an ecBufferIsEmpty error is generated.

If the ptReturnDelimiter option is on and the GetBlock call finds a delimiter in the data stream, GetBlock returns the delimiter character (and the characters preceding it) in Block. If the ptIgnoreDelim case option is on, GetBlock disregards the case of incoming characters and the case of the DelimSet characters.

Like GetChar, this is a no-wait-I/O call. GetBlock calls GetChar until it has the requested number of characters or GetChar returns an error (even if that error is ecBufferIsEmpty).

Line errors cause GetBlock to return immediately. If status buffering is enabled, the last character in the block is the one with the line error. If status buffering is disabled, you won't know which character had the line error.

Example

```

GetBlock(P, Block, 100, BytesRead, []);
case AsyncStatus of
  ecOk : {process block};
  ecBufferIsEmpty : {no characters read} ;
  ecIncompleteBlock : {process incomplete block} ;
  else {handle line error} ;
end;

```

Asks for a block of 100 received bytes (no delimiters are specified).

See Also

BlockReady
GetBlockDirect

GetBlockTimeout

GetBlockDirect / GetBlockTimeout

Syntax

```
procedure GetBlockDirect(P : PortRecPtr; var Block; ExpectedLen : Word;  
                        var ReceivedLen : Word; DelimSet : CharSet);  
procedure AbstractPort.GetBlockDirect(var Block; ExpectedLen : Word;  
                                     var ReceivedLen : Word;  
                                     DelimSet : CharSet); virtual;
```

Purpose

Read a block directly from the input buffer.

Description

This procedure is essentially identical to `GetBlock`, except that it moves the characters directly from the input buffer into `Block`, rather than calling `GetChar` repeatedly. If an unbuffered device layer is in use, this call is passed through to `GetBlock`.

Since this is generally faster than `GetBlock`, you would almost always use it instead (except, perhaps, where you are dealing with small blocks of just a few characters).

Example

```
if OutBuffUsed(P) >= 100 then  
  GetBlockDirect(P, Block, 100, BytesRead, []);
```

Read the next 100 characters from the input buffer if there are that many. When using objects, the `P` parameters would be omitted.

See Also

`GetBlock`

Syntax

```
procedure GetBlockTimeout(P : PortRecPtr; var Block; ExpectedLen : Word;  
                        var ReceivedLen : Word;  
                        DelimSet : CharSet; Timeout : Word);  
procedure AbstractPort.GetBlockTimeout(var Block; ExpectedLen : Byte;  
                                       var ReceivedLen : Word;  
                                       DelimSet : CharSet; Timeout : Word);
```

Purpose

Wait for a block or Timeout.

Description

This is a "wait-I/O" call. It is essentially identical to `GetBlock`, except that it waits up to `Timeout` clock ticks to receive the requested block.

It returns an `ecTimeout` error if the block is not received in `Timeout` ticks.

Example

```
GetBlockTimeout(P, Block, 100, BytesRead, [], 182);  
if AsyncStatus = ecTimeout then  
  Writeln('Timeout waiting for block')  
else if AsyncStatus <> 0 then  
  {handle the error} ;
```

Try for up to 10 seconds ($18.2 \times 10 = 182$) to read a block of 100 characters. When using objects, the `P` parameter would be omitted.

Syntax

```
procedure GetCharTimeout(P : PortRecPtr; var C : Char; Timeout : Word);  
procedure AbstractPort.GetCharTimeout(var C : Char; Timeout : Word);
```

Purpose

Waits for a character or Timeout.

Description

This is a "wait-I/O" call. It is identical to GetChar, except that it waits up to Timeout ticks for a received character.

It returns an ecTimeout error if Timeout clock ticks pass before the operation is completed.

Example

```
{...assume status buffering is on...}  
GetCharTimeout(P, C, 182);  
case AsyncStatus of  
  ecOk : Write(C);  
  ecTimeout : WriteLn('timed out waiting for character');  
  else {handle error} ;  
end;
```

Try for up to 10 seconds ($18.2 \times 10 = 182$) to read the next incoming character. When using objects, the P parameter would be omitted.

See Also

GetChar (core)

Syntax

```
function GetComName(P : PortRecPtr) : ComNameType;  
function AbstractPort.GetComName : ComNameType;
```

Purpose

Return the name of the specified port.

Description

This function returns the ComName for the port associated with P^. This is typically used in general purpose status routines.

Example

```
InitPortFast(P, Com4, 2400);  
...  
WriteLn('The open port is ', ComNameString(GetComName(P)));  
This call to GetComName returns Com4.  
UP.InitFast(Com4, 2400);  
...  
WriteLn('The open port is ', ComNameString(UP.GetComName));  
OOP equivalent of the previous example (UP is of type UartPort).
```

See Also

ComNameString (APPORT)

GetDelimLoc / GetLineControl

Syntax

```
function GetDelimLoc(P : PortRecPtr; DelimSet : CharSet) : Word;  
function AbstractPort.GetDelimLoc(DelimSet : CharSet) : Word;
```

Purpose

Return the location within the input buffer of the first character of DelimSet.

Description

This function can be used to determine how many characters will be returned by a call to a procedure such as GetBlock. It scans the contents of the input buffer for the specified port, looking for the first character in DelimSet. If none is found, it returns a value equal to the result of 'InBuffUsed(P)'. Otherwise it returns the distance from the head of the buffer to the delimiter character found.

GetDelimLoc is supported only for APUART (this is the only device layer in which Async Professional does the buffering).

Examples

```
(assume that input buffer contains 'ABCD;EFGH')  
GetDelimLoc(P, [';'])  
GetDelimLoc returns 5. When using objects, the P parameter would be omitted.  
GetDelimLoc(P, ['.'])  
GetDelimLoc returns 9 (the number of characters in the buffer).
```

See Also

InBuffUsed

Syntax

```
function GetLineControl(P : PortRecPtr) : Byte;  
function AbstractPort.GetLineControl : Byte; virtual;
```

Purpose

Return the line control byte from the port record.

Description

This function returns the line control register value sent to the specified port—that is, the collection of bits that was sent to the UART line control register to configure the port. It's usually of little value since GetLine provides the same information (i.e., the settings for parity, data bits, stop bits) in a more convenient form.

See Also

GetLine (core)

Syntax

```
function GetLineError(P : PortRecPtr) : Word;  
function AbstractPort.GetLineError : Word;
```

Purpose

Return the line error type (here and in AsyncStatus).

Description

This function returns the current line error, both in its function result and in AsyncStatus. Any line error conditions are then cleared (i.e., the next call to GetLineError will return ecOk).

If there is no line error pending, the result is ecOk. Other possible return values are ecOverrunError, ecParityError, and ecFramingError.

GetLineError may be called even if status buffering is turned on.

Example

```
if CheckLineError(P) then  
  WriteLn('Got line error: ', GetLineError(P));
```

Check for and report line errors. When using objects, the P parameters would be omitted.

See Also

CheckLineError

Syntax

```
function GetLineStatus(P : PortRecPtr) : Byte;  
function AbstractPort.GetLineStatus : Byte; virtual;
```

Purpose

Return the line status byte from the port record.

Description

This function returns the last value of the line status register. The value is not read from the UART but is instead returned from the specified com port record (P^). This means that the error bits and break bit are current, but the rest of the bits may be out of date.

Example

```
LSR := GetLineStatus(P);
```

This gets the last-recorded value of the line status register from port record P^. When using objects, the P parameter would be omitted.

GetModemControl / GetModemStatus

Syntax

```
function GetModemControl(P : PortRecPtr) : Byte;  
function AbstractPort.GetModemControl : Byte; virtual;
```

Purpose

Return the modem control byte from the port record.

Description

This function returns the modem control register value from the specified com port record (P^)—that is, the collection of bits that was sent to the UART modem control register to configure the handshaking lines. It's usually of little value since GetModem provides most of the same information (i.e., the settings for DTR and RTS) in a more convenient form.

See Also

GetModem (core)

Syntax

```
function GetModemStatus(P : PortRecPtr) : Byte;  
function AbstractPort.GetModemStatus : Byte;
```

Purpose

Return the modem status byte from the port record.

Description

This function returns the modem status register value from the specified com port record (P^). This value is always up to date with the actual value in the UART register.

The modem status register contains information about the modem and hardware flow control signals. If you only need to check the status of a single bit, use the individual functions (CheckCTS, CheckDCD, etc).

Example

```
MS := GetModemStatus(P);  
if (MS and DCDMask) = DCDMask then  
  WriteLn('Carrier detected');
```

Check to see if carrier was detected. When using objects, the P parameter would be omitted.

See Also

CheckCTS
CheckDCD
CheckDeltaCTS
CheckDeltaDCD

CheckDeltaDSR
CheckDeltaRI
CheckDSR
CheckRI

Syntax

```

procedure GetString(P : PortRecPtr; var S : String; ExpectedLen : Byte;
                  DelimSet : CharSet);
procedure AbstractPort.GetString(var S : String; ExpectedLen : Byte;
                               DelimSet : CharSet); virtual;

```

Purpose

Return a string with a specified length or ending with a character in DelimSet.

Description

This function returns either 1) a string of ExpectedLen characters or 2) a string ending with one of the characters in DelimSet. That is, GetString gives you a choice of terminating factors: length or a delimiting character. You must specify at least one (i.e., ExpectedLen can't be zero if DelimSet is empty); or you can specify both and the first condition to be satisfied will finish the string.

If the current device layer is buffered, the ptReturnPartialGets option is off, and neither of these conditions can be met, an ecBufferIsEmpty error is generated and S is empty. If the ptReturnPartialGets option is on, GetString returns as many characters as it can and an ecStringIncomplete error. If there are no characters to return, an ecBufferIsEmpty error is generated.

If ExpectedLen is 0 and a delimiter is not encountered within 255 characters, a string of length 255 is returned and an ecStringOverrun error is reported.

If DelimSet is not empty and a delimited string is returned, the last character in S is the delimiter if the ptReturnDelimiter option is on, as it is by default.

If DelimSet is not empty and the ptIgnoreDelimCase option is on, case is ignored in delimiter compares (ptIgnoreDelimCase is off by default).

If a character is found with a line error, GetString returns immediately and AsyncStatus has the error code. You must check for and handle line errors after the call.

Like GetChar, this is a no-wait-I/O call. GetString calls GetChar until it has the requested number of characters or GetChar returns an error (even if that error is ecBufferIsEmpty).

Examples

```

GetString(P, S, 10, []);
if AsyncStatus = ecOk then
  WriteLn('Received ', S);

```

Gets the next 10 received characters as a string. When using objects, the P parameter would be omitted.

```

GetString(P, S, 0, [cCR]);
if AsyncStatus = ecOk then
  WriteLn('Received ', S);

```

Stuffs received characters into a string until it finds a carriage return (the carriage return will be part of the string unless the ptReturnDelimiter option is off).

See Also

GetBlock

GetStringTimeout

GetStringTimeout / HWFlowDisable

Syntax

```
procedure GetStringTimeout(P : PortRecPtr; var S : String; ExpectedLen : Byte;  
                           DelimSet : CharSet; Timeout : Word);  
procedure AbstractPort.GetStringTimeout(var S : String; ExpectedLen : Byte;  
                                       DelimSet : CharSet; Timeout : Word);
```

Purpose

Waits for a string or Timeout.

Description

This is a "wait-I/O call." It is essentially identical to GetString except that it waits up to Timeout clock ticks for a received string. It returns an ecTimeout error if Timeout clock ticks pass before the operation is complete.

Example

```
GetStringTimeout(P, S, 10, [], 182);  
case AsyncStatus of  
  ecOk : WriteLn('Received ', S);  
  ecTimeout :  
    begin  
      WriteLn('Timed out waiting for string');  
      if S <> '' then  
        WriteLn('Partial string = ', S);  
    end;  
  else  
    {handle error} ;  
end;
```

Try for up to 10 seconds ($18.2 \times 10 = 182$) to read a string of 10 characters. When using objects, the P parameter would be omitted.

See Also

GetBlockTimeout

GetString

Syntax

```
{IFDEF UseHWFlow}  
procedure HWFlowDisable(P : PortRecPtr);  
procedure AbstractPort.HWFlowDisable; virtual;
```

Purpose

Disable hardware flow control.

Description

This procedure turns off hardware flow control for the specified port. This com port will no longer use the RTS and DTR modem signals to control the pacing of the remote's transmissions. And, it will no longer honor the remote's requests for transmit flow control over the DSR and CTS modem signal lines.

Example

```
HWFlowDisable(P);  
SetDTR(P, True);  
SetRTS(P, True);
```

Turns off hardware flow control and raises the DTR and RTS signals.

See Also

HWFlowEnable

HWFlowState

Syntax

```
{IFDEF UseHWFlow}
procedure HWFlowEnable(P : PortRecPtr; BufferFull, BufferResume : Word;
                      Options : Word);
procedure AbstractPort.HWFlowEnable(BufferFull, BufferResume : Word;
                      Options : Word); virtual;
```

Purpose

Enable hardware flow control (DTR and/or RTS).

Description

This procedure turns on one or both aspects of automatic hardware flow control: "receive flow control" where the kernel tells the remote to stop transmitting characters as the receive buffer fills; and "transmit flow control" where the kernel responds to the remote's request to stop transmitting characters as its input buffer fills.

Receive flow control is specified by the `hfUseDTR` and `hfUseRTS` options (defined in `APPORT`). When set, the corresponding modem control signal (DTR and/or RTS) is lowered by the Async Professional kernel when its input buffer reaches the `BufferFull` level. It is up to the remote to recognize these signals and cease sending data. There is usually little or no delay here, as there is with software flow control, so you can set the cutoff point fairly close to the size of the buffer (say at the 90% level).

As the application program removes characters from the input buffer, the buffer eventually drops below the `BufferResume` level. At that point, the kernel raises one or both of these signals. It is up to the remote to recognize this and resume sending data. Again, there should be little or no delay, so you can set the resume point fairly close to 0 (say at 10%).

Transmit flow control is specified by `hfRequireDSR` and `hfRequireCTS`. With one or both of these options set, the kernel won't transmit any data unless the remote is providing the corresponding modem status signal (DSR and/or CTS). It's up to the remote to raise and lower these signals as it needs to control the flow of transmitted characters.

The other four flow control options—`hfDTRActiveLow`, `hfRTSActiveLow`, `hfDSRActiveLow`, and `hfCTSActiveLow`—are used to tell Async Professional to invert the usual meaning of the indicated signal. For example, if a particular action is normally performed by setting the DTR signal high, that action will be performed by setting it low if the `hfDTRActiveLow` option is specified. These rarely used options are defined to allow Async Professional to be used to communicate with certain special purpose instruments. The only device layer that supports these inverted signals is `APUART`.

Example

```
InitPort(P, 2400, Com1, NoParity, 8, 1, 2000, 2000, DefPortOptions);
HWFlowEnable(P, 1800, 200, hfUseRTS+hfRequireCTS);
SetDTR(P, True);
```

After opening the com port, automatic handshaking is turned on with the cutoff point at the 90% (1800) level and the resume point at the 10% (200) level of the input buffer. Only the RTS signal will be raised or lowered as needed. The DTR signal is set permanently high.

The `hfRequireCTS` option tells the kernel to require the remote's CTS signal before transmitting (effectively enabling hardware handshaking in the transmit direction). The kernel will not require the remote's DSR signal to transmit.

```
UP.InitCustom(Com1, 2400, NoParity, 8, 1, 2000, 2000, DefPortOptions);
UP.HWFlowEnable(1800, 200, hfUseRTS+hfRequireCTS);
UP.SetDTR(True);
```

OOP equivalent of the previous example (UP is of type `UartPort`).

See Also

HWFlowDisable
HWFlowState

SetDTR
SetRTS

HWFlowState

Syntax

```
{IFDEF UseHWFlow}  
function HWFlowState(P : PortRecPtr) : FlowState;  
function AbstractPort.HWFlowState : FlowState; virtual;
```

Purpose

Return the state of hardware flow control.

Description

This function returns a value indicating the current state of hardware flow control. If flow control is off, the result is `fsOff`. If neither side of the transmission is waiting to send, the result is `fsClear`. If this side has been asked to wait, the result is `fsTransWait`. If the remote has been asked to wait, the result is `fsRecWait`. If both sides are waiting, the result is `fsAllWait`.

APINT14 always returns `fsOff` and the error `ecNotSupported`.

APFOSSIL and APDIGI14 return only `fsOff` or `fsClear`. They can report whether flow control is on, but no details beyond that.

Example

```
Write('Hardware flow control is ');  
if HWFlowState(P) <> fsOff then  
  WriteLn('ON')  
else  
  WriteLn('OFF');
```

Show the status of hardware flow control. When using objects, the P parameter would be omitted.

See Also

HWFlowDisable

HWFlowEnable

Syntax

```
function InBuffFree(P : PortRecPtr) : Word;  
function AbstractPort.InBuffFree : Word; virtual;
```

Purpose

Return the amount of free space in the input buffer.

Description

This function returns the number of free character slots in the input buffer. It returns 65535 if the device layer does not buffer input.

Example

```
repeat  
  {...get characters...}  
  {...put characters...}  
  if InBuffFree(P) < 100 then  
    PutChar(P, Xoff)  
  else if InBuffFree(P) > 400 then  
    PutChar(P, Xon);  
until Finished;
```

Implements a "poor man's flow control" from within an application's main process loop. When using objects, the P parameters would be omitted.

See Also

InBuffUsed

OutBuffFree

Syntax

```
function InBuffUsed(P : PortRecPtr) : Word;  
function AbstractPort.InBuffUsed : Word; virtual;
```

Purpose

Return the number of characters in the input buffer.

Description

This function returns the number of characters currently waiting in the input buffer. It returns 0 if the device layer does not buffer input.

Example

```
if InBuffUsed(P) > 500 then  
  GetBlockDirect(P, Block, 500, BytesRead);
```

Since this fragment makes sure there's at least 500 characters in the buffer, it doesn't need to check for partial block errors (or use a timeout call). When using objects, the P parameters would be omitted.

See Also

InBuffFree

OutBuffUsed

InitFast / InitPortFast

Syntax

```
constructor UartPort.InitFast(ComName : ComNameType; NewBaud : LongInt);
constructor FossilPort.InitFast(ComName : ComNameType; NewBaud : LongInt);
constructor Digi14Port.InitFast(ComName : ComNameType; NewBaud : LongInt);
constructor Int14Port.InitFast(ComName : ComNameType; NewBaud : LongInt);
```

Purpose

Open ComName with default line options.

Description

This constructor is the OOP equivalent of InitPortFast. It provides a convenient alternative to calling InitCustom in cases where the default line options are acceptable.

Example

```
if not UP.InitFast(Com1, 2400) then {unable to open port} ;
Opens Com1 at 2400 baud with no parity, 8 data bits, 1 stop bit, input buffer of 2048 characters,
and output buffer of 2078 characters.
```

See Also

InitPortFast

Syntax

```
procedure InitPortFast(var P : PortRecPtr; ComName : ComNameType;
                       NewBaud : LongInt);
```

Purpose

Initialize a port using default line options.

Description

This procedure provides a convenient alternative to calling InitPort in cases where the default line options are acceptable:

```
DefaultLineOptions : LineOptionRecord = (
  Parity   : NoParity;
  DataBits : 8;
  StopBits : 1;
  Options  : DefPortOptionsSimple;
  InSize   : 2048;
  OutSize  : 2048 + 30);
```

(DefaultLineOptions is a typed constant declared in APPORT.) If you don't want these defaults, you can either change this constant and continue to use InitPortFast or use InitPort.

Example

```
InitPortFast(P, Com1, 2400);
Opens Com1 at 2400 baud with no parity, 8 data bits, 1 stop bit, input buffer of 2048 characters,
and output buffer of 2078 characters.
```

See Also

InitPort (core)

XxxPort.InitFast

Syntax

```
{IFDEF UseStreams}
constructor UartPort.Load(var S : IdStream);
constructor FossilPort.Load(var S : IdStream);
constructor Digi14Port.Load(var S : IdStream);
constructor Int14Port.Load(var S : IdStream);
```

Purpose

Load a port object from a stream.

Description

This constructor loads a port object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Store.

The addresses of all user hooks (AbortFunc, ErrorProc, and WaitCharProc) must be registered with the stream before Load is called.

The stream registration routines for the port object types are UartPortStream, FossilPortStream, Digi14PortStream, and Int14PortStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
program ExLoad1; {EXLOAD1.PAS}
uses
  Crt, OpRoot, ApMisc, ApPort, ApUart, OOCOM;
var
  UP : UartPort;
  C : Char;
  Finished : Boolean;
  S : BufIdStream;
  Status : Word;

procedure Abort(Msg : String; Code : Integer);
  {-Close port and halt}
begin
  WriteLn(Msg, Code);
  Halt(1);
end;

{$F+}
procedure MyErrorProc(P : Pointer; var StatusCode : Word);
begin
  with PortRecPtr(P)^ do
    WriteLn(StatusStr(StatusCode mod 10000)+' Press <Enter>');
  ReadLn;
end;

function KbdAbort : Boolean;
  {-Default abort function}
const Escape = #$1B;
var Ch : Char;
begin
  KbdAbort := False;
  if KeyPressed then begin
    Ch := ReadKey;
```

Load

```
    if Ch = #0 then
        Ch := ReadKey
    else if Ch = Escape then
        KbdAbort := True;
    end;
end;
{$F-}

begin
    {Open the existing stream}
    if not S.Init('UPORT.STM', SOpen, 1024) then
        Abort('Failed to open stream: ', InitStatus);

    {Register the port object hierarchy}
    S.RegisterHier(UartPortStream);
    S.RegisterPointer(ptErrorProc, @MyErrorProc);
    S.RegisterPointer(ptAbortProc, @KbdAbort);
    Status := S.GetStatus;
    if Status <> 0 then
        Abort('Error registering port object: ', Status);

    {Load UP from the stream}
    S.Get(UP);
    Status := S.GetStatus;
    if Status <> 0 then
        Abort('Error loading port object: ', Status);
    S.Done;

    {Use the port in a simple terminal}
    Writeln('Press <AltX> to quit');
    Finished := False;
    repeat
        {Process chars to send}
        if KeyPressed then begin
            C := ReadKey;
            if C = #0 then begin
                C := ReadKey;
                if C = #$2D then
                    Finished := True;
            end
            else if UP.TransReady then
                UP.PutChar(C);
        end;

        {Process chars received}
        if UP.CharReady then begin
            UP.GetChar(C);
            if AsyncStatus <> ecOk then begin
                Writeln('M-J'Line error ', AsyncStatus);
                UP.FlushInBuffer;
            end else
                Write(C);
        end;
    until Finished;
    UP.Done;
end.
```


Instantiates UP from a stream created by the Store example using the stream's Get method, which calls the Load constructor indirectly. This example includes a simple terminal loop so that you can test that the Load actually did work.

OutBuffFree / OutBuffUsed

Syntax

```
function OutBuffFree(P : PortRecPtr) : Word;  
function AbstractPort.OutBuffFree : Word; virtual;
```

Purpose

Return the amount of free space in the output buffer.

Description

This function returns the amount of free space in the output buffer for the specified port. It returns 65535 if the device layer does not buffer output.

Example

```
if OutBuffFree(P) >= BlockLen then  
  PutBlockDirect(P, Block, BlockLen, BytesWritten);
```

Makes sure there's enough room in the output buffer before sending a large block of data. When using objects, the P parameters would be omitted.

See Also

InBuffFree

Syntax

```
function OutBuffUsed(P : PortRecPtr) : Word;  
function AbstractPort.OutBuffUsed : Word; virtual;
```

Purpose

Return the number of characters in the output buffer.

Description

This function returns the number of characters currently in the output buffer, waiting to be transmitted. It returns 0 if the device layer doesn't support output buffering.

Example

```
PutBlock(P, Block, 500, BytesWritten);  
while OutBuffUsed(P) > 0 do {wait};
```

Stuffs a block of data into the output buffer and then waits for that data to be completely transmitted. When using objects, the P parameters would be omitted. (Note that a better way to do this particular operation would be to use DrainOutBuffer.)

See Also

DrainOutBuffer

InBuffUsed

Syntax

```
procedure PeekCharTimeout(P : PortRecPtr; var C : Char;  
                          PeekAhead : Word; Timeout : Word);  
procedure AbstractPort.PeekCharTimeout(var C : Char; PeekAhead : Word;  
                                       Timeout : Word);
```

Purpose

Wait for a character or Timeout.

Description

This is a "wait-I/O" call. It is essentially identical to PeekChar except that it waits up to Timeout ticks for the desired characters.

PeekAhead is the number of characters in the buffer to peek ahead, with one being the character that would be returned by the next GetChar.

PeekCharTimeout returns an ecTimeout error if the character hasn't been received in Timeout ticks.

APDIGI14 can only peek ahead one character. Attempts to peek ahead more than one character generate an ecInvalidArgument error.

APINT14 and APFOSSIL cannot peek ahead, so they generate an ecNotSupported error.

Example

```
PeekCharTimeout(P, C, 1, 182);
```

Try for up to 10 seconds ($18.2 \times 10 = 182$) to peek at the next character in the input buffer. When using objects, the P parameter would be omitted.

See Also

PeekChar (core)

Syntax

```
function ProtocolInProgress(P : PortRecPtr) : Boolean;  
function AbstractPort.ProtocolInProgress : Boolean;
```

Purpose

Return True if this port is currently processing a protocol.

Description

This function returns True if the specified port is in the process of sending or receiving a file using a transfer protocol (XModem, YModem, etc.). It's primarily useful in user-written error handlers for programs that perform protocol file transfers.

Example

See the examples for SetErrorProc.

See Also

SetErrorProc

ptOptionsAreOn / ptOptionsOff / ptOptionsOn

Syntax

```
function ptOptionsAreOn(P : PortRecPtr; OptionFlags : Word) : Boolean;  
function AbstractPort.ptOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose

Return True if all specified options are on.

Description

This function returns True if the specified port option(s) are currently selected.

Example

```
if ptOptionsAreOn(P, ptIgnoreDelimCase) then  
  ptOptionsOff(P, ptIgnoreDelimCase)  
else  
  ptOptionsOn(P, ptIgnoreDelimCase);
```

Toggle the state of the ptIgnoreDelimCase option. When using objects, the P parameters would be omitted.

Syntax

```
procedure ptOptionsOff(P : PortRecPtr; OptionFlags : Word);  
procedure AbstractPort.ptOptionsOff(OptionFlags : Word);
```

Purpose

Deactivate multiple options.

Description

This procedure deactivates the specified port option(s), excluding those designated as BadPortOptions.

Example

See the example for ptOptionsAreOn.

Syntax

```
procedure ptOptionsOn(P : PortRecPtr; OptionFlags : Word);  
procedure AbstractPort.ptOptionsOn(OptionFlags : Word);
```

Purpose

Activate multiple options.

Description

This procedure activates the specified port option(s), excluding those designated as BadPortOptions.

Example

See the example for ptOptionsAreOn.

Syntax

```
procedure PutBlock(P : PortRecPtr; var Block; BlockLen : Word;
                  var BytesWritten : Word);
procedure AbstractPort.PutBlock(var Block; BlockLen : Word;
                               var BytesWritten : Word); virtual;
```

Purpose

Transmit a block of data.

Description

This procedure transmits a Block of data of length BlockLen. PutBlock normally transmits as many characters as possible (by calling PutChar repeatedly) and returns immediately if it encounters any errors. The number of characters actually transmitted is returned in BytesWritten.

If the current device layer is buffered, the ptExecutePartialPuts option is off, and there is less than BlockLen free space in the output buffer, PutBlock aborts with an ecBufferIsFull error without sending any characters.

Example

```
BlockLen := 500;
if OutBuffFree(P) > BlockLen then begin
  PutBlock(P, Block, BlockLen, BytesWritten);
  {check for errors}
end;
```

Transmit a block of 500 characters if possible.

Syntax

```
procedure PutBlockDirect(P : PortRecPtr; var Block; BlockLen : Word;
                        var BytesWritten : Word);
procedure AbstractPort.PutBlockDirect(var Block; BlockLen : Word;
                                       var BytesWritten : Word); virtual;
```

Purpose

Put a block of data directly into the output buffer.

Description

This procedure is essentially identical to PutBlock, except that it moves the characters directly from Block into the output buffer, rather than calling PutChar repeatedly. If the device layer in use is unbuffered, this call is passed through to PutBlock. If it is buffered, then the only possible error is ecBufferIsFull, which indicates that the output buffer does not have room for the Block.

Since this routine is faster than PutBlock, you would almost always want to use it instead of PutBlock (except, perhaps, where you are working with blocks of only a few characters).

Example

```
BlockLen := 500;
if OutBuffFree(P) > BlockLen then begin
  PutBlockDirect(P, Block, BlockLen, BytesWritten);
  {check for errors}
end;
```

Transmit a block of 500 characters if possible. When using objects, the P parameters would be omitted.

PutBlockTimeout / PutCharTimeout

Syntax

```
procedure PutBlockTimeout(P : PortRecPtr; var Block; BlockLen : Word;  
                           var BytesWritten : Word; Timeout : Word);  
procedure AbstractPort.PutBlockTimeout(var Block; BlockLen : Word;  
                                       var BytesWritten : Word;  
                                       Timeout : Word);
```

Purpose

Transmit a block of data or Timeout.

Description

This is a "wait-I/O" call. It is essentially identical to PutBlock except that it aborts with an ecTimeout error if Timeout clock ticks pass before the operation is completed (completed meaning that all characters were inserted into the output buffer).

Example

```
BlockLen := 500;  
PutBlockTimeout(P, Block, BlockLen, BytesWritten, 182);  
{check for errors}
```

Try for the next 10 seconds ($18.2 \times 10 = 182$) to transmit a block of 500 characters. When using objects, the P parameters would be omitted.

See Also

PutBlock

Syntax

```
procedure PutCharTimeout(P : PortRecPtr; C : Char; Timeout : Word);  
procedure AbstractPort.PutCharTimeout(C : Char; Timeout : Word);
```

Purpose

Put a character in output buffer or Timeout.

Description

This is a "wait-I/O" call. It is essentially identical to PutChar except that it waits up to Timeout ticks to send the character.

It returns an ecTimeout error if Timeout clock ticks pass before the operation is completed.

Example

```
PutChar(P, C, 182);
```

Try for up to 10 seconds ($18.2 \times 10 = 182$) to send the character in C. When using objects, the P parameter would be omitted.

See Also

PutChar (core)

Syntax

```
procedure PutString(P : PortRecPtr; S : String);  
procedure AbstractPort.PutString(S : String); virtual;
```

Purpose

Put a string directly in the output buffer.

Description

This procedure transmits the string S over the specified port.

If the device layer is buffered, and there is not room for the entire string in the output buffer, the `ecBufferIsFull` error is generated, unless the `ptExecutePartialPuts` port option is in effect, in which case as much of the string as possible is sent.

If the device layer is not buffered, then `PutString` tries to send the entire string, one character at a time. If it encounters an error at any point, it generates an `ecTransmitFailed` error.

Note that `PutString` sends only the string data (the length byte is *not* sent).

Example

```
PutString(P, 'Hello World!'+cCR);
```

Sends the string 'Hello World!' and a carriage return. When using objects, the P parameter would be omitted.

See Also

`PutBlock`

`PutChar (core)`

Syntax

```
procedure PutStringTimeout(P : PortRecPtr; S : String; Timeout : Word);  
procedure AbstractPort.PutStringTimeout(S : String; Timeout : Word);
```

Purpose

Put a string in the output buffer or Timeout.

Description

This is a "wait-I/O" call. It is essentially identical to `PutString` except that it waits up to Timeout clock ticks to transmit the string.

It returns an `ecTimeout` error if Timeout clock ticks pass before the operation is completed.

Example

```
PutString(P, 'abcd', 182);
```

Try for up to 10 seconds ($18.2 \times 10 = 182$) to send the string 'abcd'. When using objects, the P parameter would be omitted.

SetAbortFunc

Syntax

```
procedure SetAbortFunc(P : PortRecPtr; AFunc : AbortFunc);  
procedure AbstractPort.SetAbortFunc(AFunc : AbortFunc);
```

Purpose

Set AFunc as the user abort function.

Description

This routine allows you to specify a user-defined abort function. The abort function is called by all "wait I/O" Async Professional routines to give the user an opportunity to abort the current operation. It is also called by high-level file transfer functions to provide an opportunity to abort the file transfer.

The abort function should be of the form:

```
{F+}  
function AbortFunction : Boolean;  
begin  
  AbortFunction := False;  
  ...  
end;
```

It should return True to abort, False to continue. If it returns True, the current operation sets AsyncStatus to ecUserAbort.

Note that, when using OOCOM, you may override the virtual method UserAbort in lieu of calling SetAbortFunc.

By default the abort function is set to the supplied NoAbortFunc (APPORT). This function always returns False, so the process is never aborted.

For more information see "User Abort Function" in §4.B.

Example

```
{F+}  
function AbortFunction : Boolean;  
begin  
  AbortFunction := False;  
  if KeyPressed then  
    case ReadKey of  
      ^C, ^I : AbortFunction := True;      {^C or Esc}  
      #0 : AbortFunction := (ReadKey = #0); {^Break}  
    end;  
  end;  
  ...  
  SetAbortFunc(P, AbortFunction);
```

This abort function returns True if <^C>, <Esc>, or <^Break> was pressed. When using objects, the P parameter would be omitted.

Syntax

```
procedure SetDTR(P : PortRecPtr; State : Boolean);  
procedure AbstractPort.SetDTR(State : Boolean);
```

Purpose

Raise or lower DTR.

Description

This routine raises (State = True) or lowers (State = False) the DTR (Data Terminal Ready) signal. This signal does not affect the behavior of the Async Professional kernel at all; however, this signal may be required by the remote before it will transmit.

Example

```
SetDTR(P, True);  
repeat  
  GetChar(P, C);  
  {...process data...}  
until Finished;
```

Before processing received data, tell the remote to start transmitting by raising DTR. When using objects, the P parameters would be omitted.

See Also

HWFlowEnable
InitPort (core)

SetModem (core)
SetRTS

SetErrorProc

Syntax

```
procedure SetErrorProc(P : PortRecPtr; EP : AsyncErrorProc);
procedure AbstractPort.SetErrorProc(EP : AsyncErrorProc);
```

Purpose

Sets an error handler for the port.

Description

This routine allows you to install a user-written error handler, which is called by GotError in the event of an error. The error handler should be of the form:

```
{F+}
procedure ErrorProc(P : Pointer; var StatusCode : Word);
begin
  ...
end;
```

StatusCode is the Async Professional code for the error that occurred. If you are using objects, P is a pointer to a port object; if not, P is a pointer to a port record. If you have any use for the P parameter, you must typecast it to the appropriate type. StatusCode is made a VAR parameter to allow the error handler to cancel errors or change their class (e.g., from fatal to non-fatal).

One important thing to keep in mind when writing an error handler for a program that performs protocol file transfers: In the finished application, the error handler should ignore any errors that occur while the file transfer is in progress, because the protocol is responsible for handling them, not the application or the user. They are reported to the error handler only for debugging purposes—that is, you might want to display the codes for the errors that are reported during development.

Note also that *all* errors are reported to the error handler. For some applications, you may decide to ignore various errors (either during debugging or in the finished application). For example, if all data exchanges in your application are protected by block check characters, you may decide to ignore all line errors (overrun, parity, and framing). You could have your error handler ignore those errors by setting AsyncStatus to ecOK and let your application recover from them.

By default the error function is set to the supplied NoErrorProc (APPORT). This function ignores all errors.

For more information see "User Error Procedure" in §4.B.

Examples

```
{F+}
procedure MyErrorProc(P : Pointer; var StatusCode : Word);
begin
  if ProtocolInProgress(PortRecPtr(P)) or
    FaxInProgress(PortRecPtr(P)) then Exit;
  {display an error message}
end;
...
SetErrorProc(P, MyErrorProc);
```

Establish MyErrorProc as the error handler for port P. Note that the error handler does nothing if a protocol file transfer or a fax transfer is in progress.

```
{F+}
procedure MyErrorProc(P : Pointer; var StatusCode : Word);
begin
  if AbstractPortPtr(P)^.ProtocolInProgress or
    AbstractPortPtr(P)^.FaxInProgress then Exit;
  {display an error message}
end;
...
UP.SetErrorProc(MyErrorProc);
```

OOP equivalent of the previous example (UP is of type UartPort).

Syntax

```
procedure SetRTS(P : PortRecPtr; State : Boolean);  
procedure AbstractPort.SetRTS(State : Boolean);
```

Purpose

Raise or lower RTS.

Description

This routine raises (State = True) or lowers (State = False) the RTS (Request To Send) signal. This signal does not affect the behavior of the Async Professional kernel at all; however, this signal may be required by the remote before it will transmit.

Example

```
SetRTS(P);  
repeat  
  GetChar(P, C);  
  {...process data...}  
until Finished;
```

Before processing received data, tell the remote to start transmitting by raising RTS. When using objects, the P parameters would be omitted.

See Also

HWFlowEnable
InitPort (core)

SetDTR
SetModem (core)

SetWaitCharProc

Syntax

```
procedure SetWaitCharProc(WCP : WaitCharProc);  
procedure AbstractPort.SetWaitCharProc(WCP : WaitCharProc);
```

Purpose

Set a global WaitChar procedure.

Description

This routine allows you to specify a procedure to be called by WaitForChar, WaitForString, and WaitForMultiString each time a character is received. This gives the application an opportunity to display or store all characters that come in while the WaitXxxx routine is active.

If using APCOM, a wait char procedure should be of the form:

```
{F+}  
procedure WaitChar(P : PortRecPtr; C : Char);  
begin  
end;
```

If using OOCOM:

```
{F+}  
procedure WaitChar(APPtr : AbstractPortPtr; C : Char);  
begin  
end;
```

In both cases, C is the character just received. The P parameter passed by APCOM points to the port record for the port being used. The APPtr passed by OOCOM is a pointer to the port object in use.

To disable a wait char procedure, call SetWaitCharProc with NoWaitChar as a parameter. NoWaitChar is a pre-defined, do-nothing wait char routine defined in both APCOM and OOCOM.

Example

```
{F+}  
procedure MyWaitChar(P : PortRecPtr; C : Char);  
begin  
  if C = 'M' then  
    WriteLn  
  else  
    Write(C);  
end;  
...  
SetWaitCharProc(MyWaitChar);  
...  
SetWaitCharProc(NoWaitChar);
```

Establish MyWaitChar as the global wait char procedure, and later disable it.

See Also

WaitForChar
WaitForMultiString

WaitForString

Syntax

```
{IFDEF StatusBuffering}  
function StatusBuffering(P : PortRecPtr) : Boolean;  
function AbstractPort.StatusBuffering : Boolean;
```

Purpose

Return the state of status buffering.

Description

Reports the current state of status buffering (True if status buffering is on, False otherwise).

Example

```
if StatusBuffering(P) then  
  WriteLn('Status buffering is on')  
else  
  WriteLn('Status buffering is off');
```

Part of a status display routine that displays the state of status buffering. When using objects, the P parameter would be omitted.

Store

Syntax

```
{IFDEF UseStreams}  
procedure UartPort.Store(var S : IdStream);  
procedure FossilPort.Store(var S : IdStream);  
procedure Digi14Port.Store(var S : IdStream);  
procedure Int14Port.Store(var S : IdStream);
```

Purpose

Store a port object to a stream.

Description

Store writes an image of the port object to the stream S.

S is a properly initialized stream object (a stream opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The addresses of all user hooks (AbortFunc, ErrorProc, and WaitCharProc) must be registered with the stream before Load is called.

The stream registration routines for the port object types are UartPortStream, FossilPortStream, Digi14PortStream, and Int14PortStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
program ExStore1; {EXSTORE1.PAS}  
uses  
  Crt, OpRoot, ApMisc, ApPort, ApUart, OOCOM;  
var  
  UP : UartPort;  
  C : Char;  
  Finished : Boolean;  
  S : BufIdStream;  
  Status : Word;  
  
procedure Abort(Msg : String; Code : Integer);  
  {-Close port and halt}  
begin  
  WriteLn(Msg, Code);  
  Halt(1);  
end;  
  
{$F+}  
procedure MyErrorProc(P : Pointer; var StatusCode : Word);  
begin  
  with PortRecPtr(P) do  
    WriteLn(StatusStr(StatusCode mod 10000)+' Press <Enter>');  
  ReadLn;  
end;  
  
function KbdAbort : Boolean;  
  {-Default abort function}  
const  
  Escape = #$1B;  
var  
  Ch : Char;  
begin  
  KbdAbort := False;  
  if KeyPressed then begin  
    Ch := ReadKey;
```

```

    if Ch = #0 then
        Ch := ReadKey
    else if Ch = Escape then
        KbdAbort := True;
    end;
end;
{$F-}

begin
    {Open a port}
    if not UP.InitCustom(Com1, 1200, NoParity, 8, 1, 500, 500,
        DefPortOptions) then
        Abort('Failed to open port: ', AsyncStatus);

    {Set various port options}
    UP.HWFlowEnable(400, 100, hfUseRTS+hfRequireCTS);
    UP.SetErrorProc(MyErrorProc);
    UP.SetAbortFunc(KbdAbort);

    {Create a new stream}
    if not S.Init('UPORT.STM', SCreate, 1024) then
        Abort('Failed to make stream: ', InitStatus);

    {Register the port object hierarchy}
    S.RegisterHier(UartPortStream);
    S.RegisterPointer(ptErrorProc, @MyErrorProc);
    S.RegisterPointer(ptAbortProc, @KbdAbort);
    Status := S.GetStatus;
    if Status <> 0 then
        Abort('Error registering port object: ', Status);

    {Store the port}
    S.Put(UP);
    Status := S.GetStatus;
    if Status <> 0 then
        Abort('Error storing UP: ', Status);

    {Clean up}
    UP.Done;
    S.Done;
end.

```

Instantiates the port object UP and stores it in a stream using the stream's Put method, which calls the object's Store method indirectly.

See UartPort.Load for an example program that reloads this object.

SWFlowDisable

Syntax

```
{IFDEF UseSWFlow}  
procedure SWFlowDisable(P : PortRecPtr);  
procedure AbstractPort.SWFlowDisable; virtual;
```

Purpose

Disable automatic software flow control (Xon/Xoff flow control).

Description

This procedure turns off automatic software flow control for the specified port. This com port will no longer automatically send Xon/Xoff characters to control the pacing of the remote's transmissions, nor will it honor such characters received from the remote.

This procedure will not automatically issue an Xon if the remote was sent an Xoff. You can either do this manually after disabling automatic flow-control or you could avoid disabling flow control whenever an Xon is pending.

Example

```
if SWFlowState(P) <> fsClear then  
  SWFlowResume(P);  
  SWFlowDisable(P);
```

Restarts the transmission, if necessary, and turns off automatic flow control. When using objects, the P parameters would be omitted.

See Also

SWFlowEnable
SWFlowResume

SWFlowState

Syntax

```
{ $IFDEF UseSWFlow }
procedure SWFlowEnable(P : PortRecPtr; BufferFull, BufferResume : Word);
procedure AbstractPort.SWFlowEnable(BufferFull, BufferResume : Word); virtual;
```

Purpose

Enable automatic software flow control (Xon/Xoff flow control).

Description

This procedure turns on both aspects of automatic software flow control: "receive flow control" where the device layer tells the remote to stop transmitting characters; and "transmit flow control" where the device layer responds to the remote's request to stop transmitting characters as its input buffer fills.

After this call, whenever the input buffer fills to the BufferFull level, the device layer sends an Xoff character to the remote. It is up to the remote to recognize the Xoff and cease sending data. Note that there is often a delay here, so you should set the cutoff point to something below the total capacity of the buffer (say at the 75% level).

As the application program removes characters from the input buffer, the buffer eventually drops below the BufferResume level. At that point, the device layer sends an Xon character to the remote. It is up to the remote to recognize this character and resume sending data. Again, there may be a delay here, so you should set the resume point to something above zero (say at 25%).

The device layer honors the remote's Xoff/Xon requests as well. That is, if the device layer receives an Xoff, it ceases transmitting immediately. It will not start transmitting again until one of two conditions is met: 1) the remote sends an Xon character, or 2) the application program calls SWFlowResume.

Examples

```
InitPort(P, Com1, 2400, NoParity, 8, 1, 2000, 2000, DefPortOptions);
```

```
...
SWFlowEnable(P, 1500, 500);
```

After opening the com port, automatic flow control is turned on with the cutoff point at the 75% (1500) level and the resume point at the 25% (500) level of the input buffer.

```
UP.InitCustom(Com1, 2400, NoParity, 8, 1, 2000, 2000, DefPortOptions);
```

```
...
UP.SWFlowEnable(1500, 500);
```

OOP equivalent of the previous example (UP is of type UartPort).

See Also

SWFlowDisable
SWFlowEnableOpt
SWFlowResume

SWFlowSetChars
SWFlowState

SWFlowEnableOpt

Syntax

```
{ $IFDEF UseSWFlow }  
procedure SWFlowEnableOpt(P : PortRecPtr; BufferFull, BufferResume : Word;  
                          Opt : Word);  
procedure AbstractPort.SWFlowEnableOpt(BufferFull, BufferResume : Word;  
                                       Opt : Word); virtual;
```

Purpose

Enable one-way software flow control.

Description

This function is very similar in form and purpose to SWFlowEnable. The primary difference is that this procedure accepts the additional parameter Opt. Set Opt to sfTransmitFlow to enable transmit flow control. Set Opt to sfReceiveFlow to enable receive flow control. Set Opt to sfTransmitFlow + sfReceiveFlow to enable two-way flow control (same as calling SWFlowEnable).

One-way flow control is supported by APUART, APFOSSIL and APDIGI14. APINT14 doesn't support software flow control at all.

Example

```
UP.InitPort(Com1, 9600, NoParity, 8, 1, 8192, 8192, DefPortOptions);  
UP.SWFlowEnableOpt(7000, 1000, sfReceiveFlow);
```

Enables one-way software flow control where Async Professional's receiver sends an Xoff when the input buffer reaches 7000 characters and sends an Xon when the input buffer is drained below 1000.

```
UP.InitPort(Com1, 9600, NoParity, 8, 1, 8192, 8192, DefPortOptions);  
UP.SWFlowEnableOpt(0, 0, sfTransmitFlow);
```

Enables one-way software flow control where Async Professional's transmitter stops transmitting when it receives an Xoff and resumes when it receives an Xon. The BufferFull and BufferResume parameters are ignored in this case.

See Also

SWFlowEnable

Syntax

```
{$IFDEF UseSWFlow}
procedure SWFlowResume(P : PortRecPtr);
procedure AbstractPort.SWFlowResume; virtual;
```

Purpose

Force transmits to resume, even if currently blocked by Xoff.

Description

This procedure tells the device layer to ignore the last Xoff character and to resume transmitting. This may be required if the remote has somehow "forgotten" to turn our transmitter back on or the Xon character was lost due to a line error.

Example

```
repeat
  {...process data...}
  {Check for lost Xon}
  case SWFlowState(P) of
    fsTransWait, fsAllWait :
      if FirstTime then begin
        FirstTime := False;
        NewTimer(ET, 10920); {10920 = 18.2*60*10}
      end else if TimerExpired(ET) then begin
        WriteLn('Xon pending for more than 10 minutes, resuming transmits');
        SWFlowResume(P);
        FirstTime := True;
      end;
    else
      FirstTime := True;
  end;
until Finished;
```

A trap for lost Xon characters. Whenever a new Xoff is received, a timer is started. If the Xoff condition remains for more than 10 minutes (roughly 10920 clock ticks), it assumes that the Xon was lost and resumes transmitting anyway. When using objects, the P parameters would be omitted.

See Also

SWFlowState

SWFlowSetChars / SWFlowState

Syntax

```
{IFDEF UseSWFlow}  
procedure SWFlowSetChars(P : PortRecPtr; OnChar, OffChar : Char);  
procedure AbstractPort.SWFlowSetChars(OnChar, OffChar : Char); virtual;
```

Purpose

Set custom Xon/Xoff characters.

Description

This routine sets the characters to be used with automatic flow control for the specified port. These characters default to cDC1 (\$11) and cDC3 (\$13) (also called cXon and cXoff, respectively). You should call this procedure *before* enabling flow control with SWFlowEnable or SWFlowEnableOpt.

Example

```
SWFlowSetChars(P, cDC2 {#$12}, cDC3 {#$13});
```

Changes the flow control characters to the secondary Xon/Xoff characters. cDC2 and cDC3 are character constants defined in APPORT. When using objects, the P parameter would be omitted.

See Also

SWFlowEnable

SWFlowEnableOpt

Syntax

```
{IFDEF UseSWFlow}  
function SWFlowState(P : PortRecPtr) : FlowState;  
function AbstractPort.SWFlowState : FlowState; virtual;
```

Purpose

Return the current state of software flow control.

Description

This function returns a value indicating the current state of software flow control. If flow control is off, the result is fsOff. If neither side of the transmission is waiting to send, the result is fsClear. If this side has been asked to wait, the result is fsTransWait. If the remote has been asked to wait, the result is fsRecWait. If both sides are waiting, the result is fsAllWait.

Fossil and Digi14 device layers can only return fsOff and fsClear states.

The Intl4 device layer does not support flow control, and returns ecNotSupported.

Example

See the example for SWFlowResume.

See Also

SWFlowResume

Syntax

```
procedure WaitForChar(P : PortRecPtr; DelimSet : CharSet;  
                    var C : Char; Timeout : Word);  
  
procedure AbstractPort.WaitForChar(DelimSet : CharSet; var C : Char;  
                                 Timeout : Word);
```

Purpose

Wait for a character in DelimSet or Timeout.

Description

This procedure sits in a loop reading characters from the specified port until 1) a character in DelimSet is found or 2) Timeout clock ticks have elapsed. If 1) comes first, the character is returned in C. If 2) comes first, an ecTimeout error is reported. If you need to store or display the characters that come in while you are waiting, you can set up a wait char procedure by calling SetWaitCharProc.

If the pIgnoreDelimCase option is on, the character compares are case insensitive (pIgnoreDelimCase is off by default).

Example

```
WaitForChar(P, ['M', 'C', '!'], Ch, 1092);
```

Wait up to 60 seconds ($60 \times 18.2 = 1092$) to receive a carriage return, a ^C, or a !.

See Also

SetWaitCharProc

WaitForString

Syntax

```
procedure WaitForMultiString(P : PortRecPtr; SL : String;  
                           SepChar : Char; var FoundS : String;  
                           var FoundI : Byte; Timeout : Word);  
  
procedure AbstractPort.WaitForMultiString(SL : String;  
                                         SepChar : Char; var FoundS : String;  
                                         var FoundI : Byte; Timeout : Word);
```

Purpose

Wait for a substring or Timeout.

Description

This procedure sits in a loop reading characters from the specified port until 1) one of the substrings in SL is received or 2) Timeout clock ticks have elapsed. SL is a string containing multiple substrings, separated by SepChar. If 1) comes first, FoundS returns the substring that was received, and FoundI indicates which substring was received (e.g., 3 if the third substring in SL was received). If 2) comes first, an ecTimeout error is reported.

If the pIgnoreDelimCase option is set, the string compares are case insensitive.

If you need to store or display the characters that come in while you are waiting, you can set up a wait char procedure by calling SetWaitCharProc.

Example

```
WaitForMultiString(P, 'one^two^three^four', '^', FoundS, FoundI, 1092);
```

Wait for up to 60 seconds ($60 \times 18.2 = 1092$) to receive one of the four substrings. If 'three' is found, on return FoundS is 'three' and FoundI is 3.

See Also

SetWaitCharProc

WaitForString

WaitForString

Syntax

```
procedure WaitForString(P : PortRecPtr; S : String; Timeout : Word);  
procedure AbstractPort.WaitForString(S : String; Timeout : Word);
```

Purpose

Wait for a specified string or Timeout.

Description

This procedure sits in a loop reading characters from the specified port until 1) the string S is received or 2) Timeout clock ticks have elapsed. If 2) comes first, an ecTimeout error is reported.

If you need to store or display the characters that come in while you are waiting, you can set up a wait char procedure by calling SetWaitCharProc.

Example

```
WaitForString(P, 'Forum !', 1092);
```

Wait for up to 60 seconds ($60 * 18.2 = 1092$) to receive the string 'Forum !'. When using objects, the P parameter would be omitted.

See Also

SetWaitCharProc
WaitForChar

WaitForMultiString

6. Modems

A. Overview

A sizable portion of Async Professional users will be doing their serial communications using modems. Therefore, Async Professional provides a hefty add-on layer of routines in APMODEM and OOMODEM to ease the task of taming modems.

These routines build on the core and interface layers to provide easy-to-use modem control routines. You can call routines like DialModem and SetErrorControl rather than transmitting raw AT commands and deciphering the responses yourself.

You can approach APMODEM/OOMODEM from two perspectives. If you *like* issuing your own AT commands, that is made easier for you with two routines called PutModemCommand and GetModemResponse. Using these routines is much easier than working with the raw PutString/GetString methods of the interface layer (mostly because GetModemResponse interprets the modem responses for you and returns the response as an AsyncStatus code).

If you're looking for a cleaner approach, however, use the high-level routines of APMODEM/OOMODEM. Using customizable tables, these routines assemble the appropriate AT commands, send the commands, and deal with the possible responses. This means you don't have to worry about AT commands at all. Just issue the high-level call, DialModem, and check the results via AsyncStatus.

Two additional modules are provided for use when you are using the non-OOP calling format and are using one of the following modems. For a U.S. Robotics Courier 9600 BPS modem use APCOUR. For a Microcom QX/3296c 9600 BPS modem use APMCOM. These modules provide the command and response tables required for these modems.

Fundamental modem concepts are not explained in this manual. You should already know what the modem does and how you typically would use it. If you aren't familiar with these concepts, please see your modem documentation.

B. OOP vs. Non-OOP Interface

Async Professional modem support continues the split started at the interface layer. The OOP and non-OOP routines are completely separate and have separate source code files: OOMODEM for the OOP format, APMODEM for the non-OOP format.

The routines in the modem add-on layer, just like the interface layer, are nearly identical between the OOP and non-OOP formats. So again, they are documented only once, showing the appropriate declarations and pointing out the few differences. In the interface layer, the only consistent difference between the two formats was that the non-OOP format required a PortRecPtr. Similarly, the only consistent difference between the two modem routine formats is that the non-OOP format requires a ModemRecPtr.

Here are some code fragments that clarify these differences:

non-OOP:

```
var
    ComPort : PortRecPtr;
    Modem : ModemRecPtr;
...
{Instantiate a port}
InitPortFast(ComPort, Com3, 19200);
{...handle errors...}

{Instantiate a modem}
InitModem(Modem, ComPort);
{...handle errors...}

{Use word (verbal) codes}
SetModemResults(Modem, mWordResults);
{...handle errors...}
```

OOP:

```
var
    ComPort : UartPortPtr;
    Modem : HayesModemPtr;
...
{Instantiate a port}
New(ComPort, InitFast(Com3, 19200));
{...handle errors...}

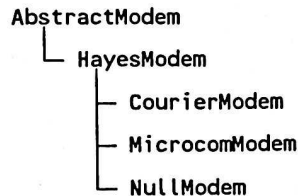
{Instantiate a modem}
New(Modem, Init(ComPort));
{...handle errors...}

{Use word (verbal) codes}
Modem^.SetModemResults(mWordResults);
{...handle errors...}
```

The non-OOP example shows a PortRecPtr (ComPort) and a ModemRecPtr (Modem). ComPort is initialized first by the call to InitPortFast. The ComPort pointer is passed to the InitModem routine which returns an initialized Modem pointer. Modem is passed to all modem routines as is shown in the call to SetModemResults.

The OOP example shows a UartPortPtr (ComPort) and a HayesModemPtr (Modem). ComPort is instantiated first and is passed to Modem's constructor. The example then commands the modem to use verbal result codes.

If you are using modem objects, here is the hierarchy you'll be working with:



AbstractModem does the majority of the work. It has the low-level routines (such as PutModemCommand and GetModemResponse) to send modem commands and get the responses. It also has all of the high-level routines (such as ResetModem and DialModem) that make working with modem objects so easy. What it *doesn't* have is any knowledge of the modem commands or possible responses. This information is provided by the derived objects in the form of tables.

In fact, the only methods implemented by derived objects are the constructors (so that the proper tables are used) and a few methods that can't be handled by the command and response tables.

The NullModem object is a special case and has no non-OOP counterpart. It overrides the PutModemCommand, GetModemResponse, and DialModem methods so that they don't really do anything at all. This allows you to test your applications by using direct connections to other PCs (via a null modem cable) without disturbing your modem control logic.

Obviously, when using the non-OOP format, you can't rely on inheritance and polymorphism to support the Courier and Microcom modems. Instead, the routines for these modem types are in separate units:

- APCOUR - routines for the U.S. Robotics Courier 9600 BPS modems
- APMCOM - routines for the Microcom QX/3296c 9600 BPS modems

These units interface special initialization routines (InitModemCourier and InitModemMcom) which set up a particular modem as either a U.S. Robotics Courier modem or a Microcom QX/3296c modem, respectively.

APCOUR and APMCOM also interface a few routines that can't be handled by the command and response tables. These routines are *not* uniquely named. For example, each unit interfaces a routine called SetFlowControl. If you are using just one of the units, this won't be an issue. If, however, you are using both units (so as to provide runtime selection of modem types), then you need to add unit qualifiers when calling these routines. For example, assume you have a variable called Modem, which is of type ModemRecPtr. You can initialize this variable to work with a Courier modem or a Microcom modem by calling the appropriate initialization routine (InitModemCourier or InitModemMCom). To call a routine from APCOUR or APMCOM, you must add a unit qualifier something like this:

```
procedure MySetFlowControl(M : ModemRecPtr; FlowOpts : Word);
begin
  case SelectedModem of
    1 : ApCour.SetFlowControl(M, FlowOpts);
    2 : ApMCom.SetFlowControl(M, FlowOpts);
  end;
end;
```

This example assumes that there is a global variable called SelectedModem that was set previously to indicate how M was initialized (with InitModemCourier or InitModemMCom). Then the case statement uses qualified calls to get the appropriate SetFlowControl.

C. Modem Control: APMODEM/OOMODEM

From a physical standpoint, the modem is a device attached to a serial port. From a logical standpoint, however, the modem is just an extension of that serial port. More than likely, the application itself won't care that the modem is there--all the application is concerned about is moving data from endpoint to endpoint. For example, consider a program running on a PC somewhere in California that needs to transfer data to a PC located in Maryland. Once there is a link between these two machines, the program proceeds (using GetChar, PutChar, and so on) without regard for the fact that the connection involves two modems and a telephone network.

The key phrase in the previous paragraph was "once there is a link." Unfortunately, the modem isn't going to do this automatically; it needs instructions. You can do this by setting its DIP switches or panel controls. However, the most common way is by sending commands from your serial port to the modem. That's where APMODEM/OOMODEM come into play.

Issuing AT Commands

When a modem is first turned on, it is in what is called "command mode." That means it tries to interpret everything it receives from the local computer as a command--a request for the modem to do something--rather than data to be passed down the phone line. Most modems these days use a superset of commands and responses from the original Hayes SmartModem 1200. Some modems also define their own command set; however, you would most likely want to use them in their "AT compatibility" modes when working with APMODEM/OOMODEM.

Without APMODEM/OOMODEM, a typical command and response cycle between your program and the modem would look something like this:

```
PutString('ATD555-1212'+cCR);
GetStringTimeout(S, 0, [cCR], Secs2Tics(20));
if (AsyncStatus <> ecOk) or (Pos('CONNECT', S) = 0) then
  {handle error}
else
  WriteLn('Connection established');
```

This example sends a command to the modem to dial the number 555-1212. It waits 20 seconds for a modem response terminated by a carriage return. It assumes the modem is returning verbal responses. If there is a line error or a timeout, or if the modem doesn't return a string containing 'CONNECT' then an error has occurred. If you wanted to find out what kind of connection was established (or what kind of error occurred) then the code following GetStringTimeout would need to be much more complex.

To make this process much easier, APMODEM and OOMODEM provide two low-level functions that encompass much of the detail work of sending modem commands and handling modem responses. Let's take a look at the same example using the modem object:

```
Modem^.PutModemCommand('ATD555-1212'+cCR);
Modem^.GetModemResponse(Secs2Tics(20));
case AsyncStatus of
  ecConnect :
    begin
      WriteLn('Connection established');
      WriteLn('BPS: ', Modem^.GetConnectSpeed);
      if Modem^.GetLastErrorMode then
        WriteLn('Error correction will be used');
    end;
  ecBusy : {handle line busy error}
  ecNoAnswer : {handle no answer error}
```

```
...
end;
```

There are several advantages to this method. First, it works whether the modem is returning verbal responses or numeric responses—it handles both automatically. Second, you can use the functions `GetConnectSpeed` and `GetLastErrorMode` rather than scan through the response string for bits per second and error control indicators. `GetModemResponse` can analyze modem responses even if the modem returns numeric responses by using an internal table to determine what each possible response means.

`APMODEM/OOMODEM` provide yet another, even higher-level approach to the same problem. They provide over 50 routines that encapsulate specific calls to `PutModemCommand` and `GetModemResponse` into easy-to-use routines like `ResetModem`, `DialModem`, `SetModemRegister`, and so on. Here's the same example using the high-level routine `DialModem`:

```
Modem^.DialModem('555-1212');
case AsyncStatus of
  ecConnect    :
    begin
      WriteLn('Connection established');
      WriteLn('BPS: ', Modem^.GetConnectSpeed);
      if Modem^.GetLastErrorMode then
        WriteLn('Error correction will be used');
      end;
    ecBusy : {handle line busy error}
    ecNoAnswer : {handle no answer error}
  ...
end;
```

The `Put/Get` routines are replaced with a call to `DialModem`. The first thing this does is to clean up the source code a bit. It might not be obvious what 'ATD555-1212' means, but `DialModem` is pretty clear. (This is probably a bad example since ATD555-1212 is easy to figure out, but a high-level routine named `SetModemSpeaker` is probably easier to figure out than 'ATM1'.)

However, these routines do much more than just clean up the source code. By using command and response tables, the `DialModem` routine and all other high-level routines can work with any modem regardless of the actual command strings and responses used by the modem.

Command and Response Tables

Every modem object (and modem record, if using `APMODEM`) has three tables associated with it: a command table, a verbal response table, and a numeric response table. The command table is used by all of the high-level routines to look up the appropriate command string for the task. The two response tables are consulted by `GetModemResponse` in order to interpret the modem's response.

`APMODEM` and `OOMODEM` supply three sets of these tables: a set for Hayes and Hayes-compatible modems, a set for U.S. Robotics Courier 9600 BPS modems, and a set for Microcom QX/3296c 9600 BPS modems. Initialize your modem object as one of these types and, from then on, your modem object will use the appropriate tables.

The Hayes tables contain the almost universally standard AT commands supported by nearly all 1200 and 2400 BPS modems available today. You can use these tables for nearly all 1200 and 2400 baud non-MNP, non-V.42 modems. You could also use them for MNP or V.42 modems, but you'd need to add command table entries for the extended commands; you might need to add new entries to the response tables as well. The Courier and Microcom tables contain the standard and extended command strings, and possible responses, for those modems.

Here's an extract from OOMODEM that shows what these three tables look like:

```
{Basic Hayes command set}
HayesCommandMax = 110;
HayesCommandID : String[14] = 'hayes commands';
HayesCommandSet : array[0..HayesCommandMax] of Byte = (
  (len flags chars      command type      modem command)
  3, $07, cA,            mcAnswer,        {Answer phone}
  4, $00, cA, Ord('/'),  mcRepeat,      {Repeat cmd}
  3, $03, cC,            mcSetCarrierTrans, {Trans on/off}
  3, $0F, cD,            mcDial,           {Dial number string}
  3, $03, cE,            mcEcho,           {Set offline echo}
  ...

{Modem response text set}
HayesResponseMax = 120;
HayesResponseID : String[14] = 'hayes responses';
HayesResponseSet : array[0..HayesResponseMax] of Byte = (
  (len text fragment      response code      meaning)
  8, cC,cO,cN,cN,cE,cC,cT, mrConnect,      {Connect}
  11, cN,cO,c_,cC,cA,cR,cR,cI,cE,cR, mrNoCarrier,    {No carrier}
  6, cE,cR,cR,cO,cR,      mrError,          {Error}
  ...

{Modem response code set}
HayesCodeMax = 60;
HayesCodeID : String[11] = 'hayes codes';
HayesCodeSet : array[0..HayesCodeMax] of Byte = (
  (modem response options response code      meaning)
  0, $00,$00, mrOk,          {Ok}
  1, $00,$02, mrConnect,     {Connect}
  2, $00,$00, mrRing,        {Ring}
  3, $00,$00, mrNoCarrier,    {No carrier}
  ...
```

The first table, HayesCommandSet, is used by all high-level routines (DialModem, ResetModem, etc.). In fact, the majority of the routines in APMODEM/OOMODEM are nothing more than shells around a call to ExecuteModemCommand. For example:

```
procedure AbstractModem.ResetModem;
  {-Issues ATZ command to reset modem to power-on defaults}
begin
  ExecuteModemCommand(mcReset, -1);
end;
```

Here, ExecuteModemCommand is passed the command code of mcReset. It looks up mcReset in the command table and extracts the associated command string (in this case, just a 'Z'). It adds the AT prefix and the carriage return suffix and sends that command (via PutModemCommand).

ExecuteModemCommand also handles the modem response by calling GetModemResponse. GetModemResponse looks up the returned string (or number) in the appropriate response table and sets AsyncStatus, the current BPS rate, and the current error control mode.

You don't need to take any specific action to get your modem object (or modem record) hooked up with the proper modem tables. This happens automatically when you initialize your modem object. Here's what the OOP initialization sequence looks like:

```

var
  ComPort : UartPort;
  Modem   : HayesModem;

...
ComPort.InitFast(Com1, 2048, 2048);
Modem.Init(@ComPort);

```

This example shows the use of a static UartPort object and a static HayesModem object. The Init constructor for the HayesModem object selects the proper tables for that modem. This assumes that this program will always be using a Hayes modem. Here's a more flexible approach:

```

var
  ComPort : UartPort;
  Modem    : AbstractModemPtr;

...
ComPort.InitFast(Com1, 2400);
case SelectedModem of
  1 : Modem := New(HayesModemPtr, Init(@ComPort));
  2 : Modem := New(CourierModemPtr, Init(@ComPort));
  3 : Modem := New(MicrocomModemPtr, Init(@ComPort));
end;

```

This example assumes that the identifier SelectedModem is set at runtime to indicate the attached modem type. Because Modem is of type AbstractModemPtr, it can point to any type of derived modem. The case statement calls whatever constructor is appropriate for the selected modem.

You can do much the same thing with the non-OOP interface:

```

var
  P : PortRecPtr;
  Modem : ModemRecPtr;

...
InitPortFast(P, Com1, 2400);
case SelectedModem of
  1 : InitModem(Modem, P);
  2 : InitModemCourier(Modem, P);
  3 : InitModemMcom(Modem, P);
end;

```

After using any of these examples, you can proceed to call the many high-level routines of APMODEM/OOMODEM without regard to the attached modem type. The proper AT commands will always be issued and the returned response will be correctly interpreted.

Modifying the Tables

In some cases, you may need to modify all or part of a command or response table. There are several ways to do this. You can modify the tables in APMODEM and OOMODEM. Or, you can create your own table, starting from one of the built-in ones, and have your modem object use that table instead of the built-in ones by calling the methods SetModemCmdTable, SetModemRespTable, and SetModemCodeTable. Or finally, you can make small adjustments in the built-in tables at runtime by calling the methods described next.

For the command table, use the methods AddModemCommand and RemoveModemCommand to make modifications. There are several types of changes these routines can handle: changing an existing command string, removing an existing command string, or adding a new user command and associated command string. The following discusses each in turn.

Suppose you have a modem that uses 'ATB' to set modem volume instead of the more standard 'ATL'. You could change this at runtime as follows:

```
AddModemCommand(mcVolume, 'B', moUsePrefix+moUseSuffix);
```

This causes SetModemVolume to assemble and send the following command string:

```
ATBn<CR> with n being the number passed to SetModemVolume
```

Suppose you simply want to remove a command. Assume your modem doesn't support the volume command (not all modems do). If you were to send this command to the modem, it would return an error response. You can avoid the modem error by removing this command from the command table:

```
RemoveModemCommand(mcVolume);
```

Now a call to SetModemVolume generates the error ecNullCommand, meaning that no command string exists for that command. Your program still has to deal with an error, but it's doing it internally, without wasting time going to the modem.

Perhaps the most likely modification you'll need to make is to add a new command—one that isn't part of the standard or extended command sets. For example, some overseas phone systems require the modem to send something called a guard tone when answering a call. This isn't part of the standard command set, but it's easy enough to add.

```
AddModemCommand(mcUser0, '&G', moUsePrefix+moUseSuffix);
```

This adds a new command, mcUser0, and a new command string, '&G', to the current command table. You can execute this command with the following:

```
ExecuteModemCommand(mcUser0, 1);
```

ExecuteModemCommand looks up mcUser0 in the command table and finds the string '&G'. The moUsePrefix and moUseSuffix options tell it to assemble and send the following command string:

```
AT&G1<CR>
```

You might also need to modify the verbal response table (the table used by GetModemResponse to handle verbal responses). You have the same three options: modifying the supplied source code, creating and using your own tables, or modifying the standard tables at runtime with AddModemResponse and RemoveModemResponse.

An important difference from modifying command tables is that, unlike AddModemCommand, AddModemResponse never changes an existing response. The rationale behind this behavior is that there is no harm in having multiple responses that return the same response code. In fact, this may even be desirable, since it would allow you to use the same modem response table for two modems that have slightly different responses.

This behavior is used in the supplied standard modem response table, which handles two different response strings for the "no dialtone" condition. Although most modems return 'NO DIAL', there is one modem that returns 'NO TONE'. By putting both strings in the standard table, it can handle either response string.

Other than that, modifying the verbal response table works much like modifying the command table. Let's run through an example to see it in action. Assume that instead of returning 'BUSY', your modem returns 'LINE BUSY' when it finds a busy line. The simplest approach is to leave the existing 'BUSY' entry alone and add a new one to handle 'LINE BUSY':

```
AddModemResponse('LINE BUSY', mrBusy);
```

After this, the modem returns the response code mrBusy whenever it finds the string 'LINE BUSY'. See the Declarations later in this section for a complete list of possible response codes.

The third and final table that you might need to modify is the numeric code response table. `GetModemResponse` uses this table to look up numeric responses. The main purpose of this table is to return a standard response code (`mrConnect`, `mrBusy`, etc.); it also returns the BPS rate and error control information for `mrConnect` responses.

The most likely reason for modifying this table is to add responses specific to your modem (responses that are not part of the Hayes standard). For example, suppose your modem returns the number 11 when it gets an error-correcting connection at 2400 BPS. You would add this as follows:

```
AddModemCode(11, mrConnect, mn2400BPS+mnErrorControl);
```

Then your modem object would correctly identify the numeric response of 11 to mean a connection at 2400 BPS with error control. See the Declarations later in this section for a list of all numeric response options.

Modem Configuration: Two Approaches

Typically, your application will need to configure the modem to a desired state (e.g., by turning the speaker on and requesting data compression). Much of this information could be specified once and saved in the modem's non-volatile memory (obviating the need for your application to do anything except issue a dial command). This might even be a reasonable approach if you've written an application for yourself and this application always uses the modem the same way. Unfortunately, things are rarely this easy. More likely, you'll be distributing your application to others and you'll need to provide for different modem configurations.

In this case, the only reasonable thing to do is to issue the appropriate configuration commands every time the program starts. You nearly always need to do this in a "user controlled" fashion (for example, you would want the user to be able to specify whether data compression should be on or off for a particular session). You also need to deal with the situation where the user has asked for a feature that the modem doesn't support.

`APMODEM` and `OOMODEM` provide two approaches to this problem. The first, more traditional, way of doing this is to store various AT command strings within your application. You give your user the opportunity to edit these strings to turn various options on or off. This gives the user the burden of knowing what the correct AT command strings are. In some cases, this might be the right thing to do. You could take this approach by using only two routines: `PutModemCommand` and `GetModemResponse`. You would store a bunch of setup strings, let the user modify them, and then send them using `PutModemCommand`. `GetModemResponse` would report whether or not there were any errors in the command strings.

In many cases, you might want a more foolproof way of handling modem configuration. For example, you might want to request the configuration information from the user in plain English terms like "Data compression [On/Off]?". This means your program must be smart enough to know what AT commands are required for turning data compression on and off. And it needs to know this information for all the different modems you might be working with.

`OOMODEM`, in conjunction with the Object Professional stream manager, provides a way of doing this. First, you would build a library of all modem types you are likely to be working with. (Currently, `OOMODEM` provides for only three: standard Hayes, U.S. Robotics Courier 9600 BPS modems, and Microcom QX/3296c 9600 BPS modems.) Then, your application presents a list of these modem types to the user and loads the selected modem type from the library. Once the proper modem type is instantiated, the rest of your application remains the same. See `MODLIB.LZH` in the bonus library for sample programs that create and use a modem library.

Handling Errors

APMODEM and OOMODEM follow the same error handling rules as the rest of Async Professional: all routines set AsyncStatus to indicate their results. If an error occurs, the error is reported to the user error procedure. However, a few words are appropriate here about what kinds of modem errors you might receive and what you should do about them. Here are the modem-specific error codes:

```
ecUnknownModemResult  
ecTableFull  
ecNullCommand
```

ecUnknownModemResult means that GetModemResponse got a response that didn't conform to any known response format or wasn't found in the response tables. This would happen if your modem returned a response like 'NO TONE' but the modem response table had only 'NO DIALTONE' as an expected response. It could also mean that you've exercised a modem option that doesn't fall into the normal category of responses (like asking your modem to send its help screen). Since this is typically a configuration or operational error, it's more likely to show up during development than with a finished application.

ecTableFull occurs only when you are adding commands or responses to existing tables. It's telling you that there isn't enough room to add your new entry. You need to remove some existing entries or expand the table by modifying the source code.

ecNullCommand means that you requested an extended feature that the current modem type (Hayes, Courier, or Microcom) doesn't support. What you should do in such a case depends on your application. You may find it perfectly acceptable to ignore this error and continue. Or you might consider it a configuration mistake and alert the user to the fact that the operation failed.

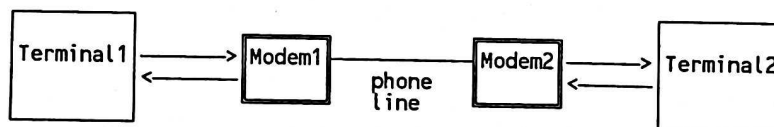
Let's consider an example. Suppose you allow your user to decide whether to activate data compression for a particular session. Further suppose that the user decides to turn data compression on *but* is using a modem that doesn't support it. Your application calls the SetDataCompression routine and it, of course, returns ecNullCommand. Since data compression isn't critical to the application, you may decide to do nothing. No action is required because the connection will still be made and data will still be transferred; it just won't be compressed. In fact, this is no different from the user asking for data compression and then calling a remote modem that doesn't support it.

On the other hand, you might decide that, harmless as this error is, it's still prudent to report the condition to your user before continuing. The decision is up to you.

Modem Flow Control

Before reading this section, be sure to read and understand the general concepts of flow control described in §5.C. This section will focus on how flow control relates to various modem configurations.

The following diagram will be used to discuss modem flow control:



Terminal1 and Terminal2 are PCs. Terminal1 is local; Terminal2 is at a distant location. Modem1 is the local modem sitting next to the local PC, Modem2 is the remote modem sitting next to the

remote PC. (In terms of the RS-232 naming conventions, the terminals are DTEs, data terminal equipment, and the modems are DCEs, data communications equipment.)

Let's take the simplest case first. Assume that the two modems, Modem1 and Modem2, are both low-speed (300 to 2400), non-MNP, non-V.42 modems. Such modems typically do not support any type of flow control between the terminal and the modem. The link between two such modems is not being managed in any way (for error control or data compression). When a serial stream of bits leaves Terminal1, it travels through Modem1, across the phone line into Modem2, and into Terminal2. So logically, this is a straight line connection and there aren't any "stopping points" where flow control might be needed.

This is not to say that you'd never have flow control in such situations. Certainly you can't have hardware flow control, since there's no direct connection between the modem control signals from Terminal1 and Terminal2. But you can, and sometimes must, have software flow control. The software flow control acts as though the modems aren't part of the connection at all—the flow control is between Terminal1 and Terminal2.

Let's say that Terminal1 is sending lots of text to the person sitting in front of Terminal2. Once the Terminal2 screen fills, that person will want to pause the flow of data while reading the text. If both terminals are honoring software flow control, then pressing <CtrlS> on the keyboard is sufficient. (<CtrlS> is actually an Xoff.) This Xoff gets transmitted without interference all the way to Terminal1, which then stops transmitting. When the person in front of Terminal2 is ready for additional text, <CtrlQ> (which is actually an Xon) is pressed. This Xon gets transmitted to Terminal1, which resumes sending text. This is software flow control.

This same flow control mechanism could be employed during large automatic transfers of data, as occurs during protocol transfers. Such automatic software flow control is part of the APUART device layer of Async Professional and therefore, part of its protocols as well.

Now let's talk about managed modem links: MNP and V.42. "Managed modem link" means that the link between Terminal1 and Terminal2 is no longer a "pass-through" connection. Instead, Modem1 collects a group of bytes from Terminal1, packages them into a block, and sends that block, complete with error control information, to Modem2. If the block is received without errors, Modem2 acknowledges receipt of the block (i.e., it tells Modem1 that it got the block OK) and passes the stream of bytes on to Terminal2. If there are errors, Modem2 *won't* pass the data on to Terminal2, but instead asks Terminal1 to retransmit the block.

Before this can work, both modems must agree to the same management scheme. Currently, the modem industry supports two standards for managing this link: MNP (for Microcom Network Protocol) and V.42 LAPM (Link Access Procedure for Modems), a standard supported by CCITT). Describing either of these standards is far beyond the scope of this manual. If your modem supports one of these standards, then your modem manual will provide the necessary information.

In short, each of these standards provides a managed link between the two modems. This provides opportunities for error detection and correction, data compression, and end-to-end hardware or software flow control.

The following example clarifies what is meant by end-to-end hardware flow control. Terminal1 is transmitting data at a high rate over the link to Terminal2. Terminal2 can't process the data fast enough, so it drops its RTS line, forcing Modem2 to stop passing data. This also causes Modem2 to stop accepting data from Modem1. Since Modem1 isn't accepting data any more, it lowers its CTS signal, telling Terminal1 to stop transmitting its data. And there you have it—end-to-end hardware flow control.

You can also use software flow control in these cases. Unlike an unmanaged link, however, the software flow control is between the terminal and modem instead of between the two terminals. But, as we just saw above, a managed link's flow control between the terminal and modem is just as good as terminal to terminal in an unmanaged link.

Managed links generally *require* some type of flow control—either software or hardware. This is true even if the link is operating at relatively low speeds (1200 or 2400 BPS) since you never know when modem-to-modem retransmissions might occur, potentially causing the transmitting terminal to overflow its modem's buffer.

Shared Declarations

Constants

```
cA = $41;  
cB = $42;  
cC = $43;  
cD = $44;  
cE = $45;  
cF = $46;  
cG = $47;  
cH = $48;  
cI = $49;  
cJ = $4A;  
cK = $4B;  
cL = $4C;  
cM = $4D;  
cN = $4E;  
cO = $4F;  
cP = $50;  
cQ = $51;  
cR = $52;  
cS = $53;  
cT = $54;  
cU = $55;  
cV = $56;  
cW = $57;  
cX = $58;  
cY = $59;  
cZ = $5A;  
c_ = $20; { ' ' }
```

Convenient constants for building response strings in a modem response table. Intended primarily for internal use.

```
DefDelayFactor : Byte = 2;
```

Default time (in clock ticks) to delay between modem commands. The delay is used to avoid problems involving slow modems on fast machines. Some modems need a very brief recovery period between replying to the last command and accepting a new command. If a very fast machine is sending out modem commands back-to-back, it might send out the next command before the modem is ready. The default delay of 2 clock ticks is adequate for most machine/modem configurations.

```
DefDialTimeout : Word = 1092;
```

Default time (in clock ticks) to wait for a response to an mcDial or mcAnswer command (1092 ticks = 60 seconds).

DefSepChar : Char = '/';

Default separator character in modem response strings. This character is used internally when interpreting response strings returned by certain modems (e.g., the Courier modem, which returns strings like 'CONNECT 9600/ARQ' rather than 'CONNECT 9600').

DefTimeout : Word = 182;

Default time (in clock ticks) to wait for a response to a modem command other than mcDial and mcAnswer (182 ticks = 10 secs).

DTRDropHold : Word = 8;

The number of ticks that HangupModem holds the DTR line low. This value should be sufficient for most modems in their normal configuration. If your modem doesn't hang up using this value, you should increase DTRDropHold.

```
fSWTrans = $0001; {Software transmit flow}
fSWRec   = $0002; {Software receive flow}
fHWTrans = $0004; {Hardware transmit flow}
fHWRec   = $0008; {Hardware receive flow}
fRmtXoff = $0010; {Pass Xon/Xoff to remote}
```

Flow control option flags. See the entry for SetFlowControl for details.

HayesCodeMax = 60;

HayesCodeID : String[11] = 'hayes codes';

HayesCodeSet : array[0..HayesCodeMax] of Byte = (...);

HayesCodeSet is a table used to translate numeric responses returned by Hayes-compatible modems into APMODEM/OOMODEM response codes. HayesCodeMax is the last valid index into the table. HayesCodeID is an ID string used to mark the beginning of the table. (Note that OOMODEM/APCOUR/APMCOM provide similar tables, CourierCodeSet and MicrocomCodeSet, for U.S. Robotics Courier and Microcom 9600 BPS modems.)

HayesCommandMax = 110;

HayesCommandID : String[14] = 'hayes commands';

HayesCommandSet : array[0..HayesCommandMax] of Byte = (...);

HayesCommandSet is a table used to translate APMODEM/OOMODEM command codes into Hayes-compatible command strings. HayesCommandMax is the last valid index into the table. HayesCommandID is an ID string used to mark the beginning of the table. (Note that OOMODEM/APCOUR/APMCOM provide similar tables, CourierCommandSet and MicrocomCommandSet, for U.S. Robotics Courier and Microcom 9600 BPS modems.)

HayesResponseMax = 120;

HayesResponseID : String[15] = 'hayes responses';

HayesResponseSet : array[0..HayesResponseMax] of Byte = (...);

HayesResponseSet is a table used to translate response strings returned by Hayes-compatible modems into APMODEM/OOMODEM response codes. HayesResponseMax is the last valid index into the table. HayesResponseID is an ID string used to mark the beginning of the table. (Note that OOMODEM/APCOUR/APMCOM provide similar tables, CourierResponseSet and MicrocomResponseSet, for U.S. Robotics Courier and Microcom 9600 BPS modems.)

MaxDialStr = 30;

Maximum length of prefix for dial string (see PrefixStr type and SetDialPrefix).

MaxSRegs = 40;

Maximum number of S-registers supported.

MaxStdSRegs = 12;

Highest standard S-register number on a Hayes-compatible modem.

MaxWordLen = 30;

Maximum length for a modem response string (a word rather than a number).

```

mcNone           = 0;  {Null command}
mcAnswer         = 1;  {A - answer phone immediately}
mcRepeat         = 2;  {A/ - repeat last command}
mcSetCarrierTrans = 3;  {Cn - carrier transmitter on/off}
mcDial           = 4;  {Ds - dial number string}
mcEcho           = 5;  {En - set offline echo on/off}
mcOnlineEcho     = 6;  {Fn - set online echo on/off}
mcHook           = 7;  {Hn - set hook mode on/off/special}
mcSpeaker        = 8;  {Mn - set speaker off, on until CD, always on}
mcOnline         = 9;  {O - go online}
mcQuiet          = 10; {Q - set result mode quiet/normal}
mcSetRegister    = 11; {Sn=x - set an S register}
mcReadRegister   = 12; {Sn= - return an S register}
mcPulse          = 13; {P - pulse dialing}
mcResultCodes    = 14; {Vn - set results codes or verbal}
mcCodeSet        = 15; {Xn - set result code set}
mcReset          = 16; {Z - reset modem}
mcTone           = 17; {T - tone dialing}
mcVolume         = 18; {L - volume control, non-standard, but common}
mcDCDControl     = 19; {&C - DCD always on/follow connect}
mcDTRControl     = 20; {&D - DTR override/terminate call}
mcCmdMode        = 21; {+++ - escape to command mode}
mcDataCompression = 22; {varies with modem}
mcErrorControlOff = 23; {varies with modem}
mcErrorControlOn  = 24; {varies with modem}
mcErrorControlAuto = 25; {varies with modem}
mcLinkLockedOn    = 26; {varies with modem}
mcLinkLockedOff   = 27; {varies with modem}
mcUser0           = 100; {...user added commands...}
mcUser1           = 101;
...
mcUser9           = 109;

```

Codes for standard and user-defined modem commands. mcAnswer through mcTone correspond to the standard Hayes command set. mcVolume is a command available on many, but not all Hayes-compatible modems. mcDCDControl through mcLinkLockedOff are used only for Courier and Microcom modems. mcUser0 through mcUser9 are codes reserved for user-defined commands (see the entry for AddModemCommand).

```

mCodeResults = 0;
mWordResults = 1;

```

Function codes passed to SetModemResults.

```

mEchoOff = 0;
mEchoOn  = 1;

```

Function codes passed to SetModemEcho and SetModemOnlineEcho.

```

mExtSet0 = 0;
mExtSet1 = 1;
mExtSet2 = 2;
mExtSet3 = 3;
mExtSet4 = 4;
mExtSet5 = 5;
mExtSet6 = 6;
mExtSet7 = 7;

```

Function codes passed to SetModemCodeSet.

```

mHangup = 0;
mOnHook = 0;
mOffHook = 1;

```

Function codes passed to HangupModem.

```

mLow = 0;
mMedium = 1;
mHigh = 2;

```

Function codes passed to SetModemVolume.

```

mNormalResp = 0;
mQuietResp = 1;

```

Function codes passed to SetModemQuiet.

```

mnErrorControl = $0001; {An error correcting connection was made}
mnOtherBPS = $0002; {Other BPS connection}
mn600BPS = $0004; {600 BPS connection}
mn1200BPS = $0008; {1200 BPS connection}
mn2400BPS = $0010; {2400 BPS connection}
mn4800BPS = $0020; {4800 BPS connection}
mn7200BPS = $0040; {7200 BPS connection}
mn9600BPS = $0080; {9600 BPS connection}
mn12000BPS = $0100; {12000 BPS connection}
mn14400BPS = $0200; {14400 BPS connection}

```

Modem numeric response options. mnErrorControl means the numeric response indicates an error control connection. mnOtherBPS indicates a connection of an unknown BPS rate. mn600BPS through mn14400BPS indicate a connection of the specified BPS rate.

```

moUsePrefix = $0001; {Attach CmdPrefix 'AT' to front of command string}
moUseSuffix = $0002; {Append CmdSuffix <CR> to end of command}
moDialTimeout = $0004; {Use dial timeout value}

```

Modem command options. moUsePrefix indicates that the command should be preceded by the standard prefix (this defaults to the string AT but can be changed by directly modifying the string field CmdPrefix). moUseSuffix indicates that the command should be followed by a suffix (this defaults to a carriage return but can be changed by directly modifying the character field CmdSuffix). moDialTimeout indicates that, when waiting for a response from the modem, the dial timeout value (default = 60 seconds) should be used, rather than the shorter command timeout value (default = 10 seconds). See the entry for AddModemCommand.

```

mrOk = 0; {OK}
mrConnect = 1; {CONNECT, CONNECT 1200, CONNECT 2400, etc.}
mrRing = 2; {RING, RINGING}
mrNoCarrier = 3; {NO CARRIER}
mrError = 4; {ERROR}
mrNoDialtone = 6; {NO DIALTONE, NO DIAL TONE, NO TONE}
mrBusy = 7; {BUSY}
mrNoAnswer = 8; {NO ANSWER}
mrVoice = 9; {VOICE}
mrNone = 99;

```

Codes for standard responses from a Hayes-compatible modem.

```

mSpeakerOff = 0;
mSpeakerOnDial = 1;
mSpeakerOn = 2;
mSpeakerOnConnect = 3;

```

Function codes passed to SetModemSpeaker.

```

mTransOff = 0;
mTransOn = 1;

```

Function codes passed to SetCarrierTrans.

```
mDCDAwaysOn      = 0;
mDCDFollowConnect = 1;
```

Options passed to SetDCDControl.

```
mDTRAlwaysOn      = 0;
mDTRTerminateCall = 2;
```

Options passed to SetDTRControl.

```
SRegsInit : SRegSet = (
  (Lo:0; Hi:255; Def:0), {0 Rings to answer}
  (Lo:0; Hi:255; Def:0), {1 Ring counter}
  (Lo:0; Hi:127; Def:43), {2 Escape code (ASCII 0-127)}
  (Lo:0; Hi:127; Def:13), {3 Carriage return (ASCII 0-127)}
  (Lo:0; Hi:127; Def:10), {4 Line feed (ASCII 0-127)}
  (Lo:0; Hi:32; Def:8),   {5 Backspace (ASCII 0-33)}
  (Lo:0; Hi:255; Def:2),  {6 Wait period for dialtone}
  (Lo:1; Hi:255; Def:30), {7 Wait for ring-back/carrier}
  (Lo:0; Hi:255; Def:2),  {8 Comma pause time}
  (Lo:1; Hi:255; Def:6),  {9 Carrier recognize wait}
  (Lo:1; Hi:255; Def:7),  {10 Carrier loss wait}
  (Lo:1; Hi:255; Def:70), {11 Reserved}
  (Lo:20; Hi:255; Def:50)); {12 Escape code guard time}
```

Table containing the range of legal values for the 13 standard S-registers, and the defaults. This table can be used when calling SetModemRegister to reset a register to its default value. See the entry for SetModemRegister.

Types

```
ErrorStates = (eCheckOff, eCheckOn, eCheckAuto);
```

Error checking options. See the entry for SetErrorControl.

```
ModemCodeTable = array[0..65534] of Byte;
ModemCodePtr = ^ModemCodeTable;
```

A table of modem numeric code responses.

```
ModemCommandTable = array[0..65534] of Byte;
ModemCommandPtr = ^ModemCommandTable;
```

A table of modem commands.

```
ModemRegType = record
  Lo : byte;
  Hi : byte;
  Def : byte;
end;
```

Defines a single S-register. Lo and Hi represent the range of values that the register may contain. Def is the default value for the register.

```
ModemResponseTable = array[0..65534] of Byte;
ModemResponsePtr = ^ModemResponseTable;
```

A table of modem response strings.

```
NumberStr = String[20];
```

Maximum length of a telephone number passed to DialModem.

```
PrefixStr = String[MaxDialStr];
```

A string added to the modem dial command ('ATD') before the phone number. See the entry for SetDialPrefix.

```
ResponseType = (NumericCodes, WordCodes);
```

Type of an internal variable used to indicate whether the modem in use is configured to supply responses in the form of numbers or words.

```
SRegSet = array[0..MaxStdSRegs] of ModemRegType;
```

A table describing the legal and default values for the standard S-registers.

APMODEM Declarations

Types

```
ModemRecPtr = ^ModemRec;  
ModemRec =  
    record  
        ...  
    end;
```

A record structure used internally to store information about a modem. An application program generally works only with ModemRecPtrs.

OOMODEM Declarations

Types

```
AbstractModemPtr = ^AbstractModem;  
AbstractModem =  
    object(Root)  
        ...  
    end;
```

Abstract object that defines data fields and methods to be shared in common by higher-level modem objects.

```
CourierModemPtr = ^CourierModem;  
CourierModem =  
    object(HayesModem)  
    end;
```

Object used when working with a U.S. Robotics Courier modem.

```
HayesModemPtr = ^HayesModem;  
HayesModem =  
    object(AbstractModem)  
    end;
```

Object used when working with most Hayes-compatible modems.

```
MicrocomModemPtr = ^MicrocomModem;  
MicrocomModem =  
    object(HayesModem)  
    end;
```

Object used when working with a Microcom modem that implements the MNP error correcting protocol.

```
NullModemPtr = ^NullModem;  
NullModem =  
    object(HayesModem)  
    end;
```

Object used when working with a null modem cable between two directly connected PCs.

AddModemCode

Syntax

```
procedure AddModemCode(M : ModemRecPtr; NC, Code : Byte; Options : Word);  
procedure AbstractModem.AddModemCode(NC, Code : Byte; Options : Word);
```

Purpose

Add a numeric response to the numeric code response table.

Description

This routine is used to add a new numeric code response in the specified modem's code response table. If there is not enough room in the table for the new code, an `ecTableFull` error is generated. NC is the new numeric response code, the value returned by the modem, to be added to the table. Code is the response code that Async Professional should associate with NC (`mrOk` through `mrNone`). Note that `mrNone` is reserved for internal use by `RemoveModemCode`. Assigning a response code to `mrNone` does nothing.

Options is a bit-mapped word that indicates what BPS rate is specified by NC and whether or not this connection employs error control. If the numeric code does not indicate a connection, then Options should always be zero. `mnErrorControl` means that the connection does use error control. `mnOtherBPS` means that the BPS rate of this connection isn't in the Async Professional standard list or isn't known. `mn600BPS` through `mn14400BPS` indicate the exact BPS rate of this connection. See the numeric response options in the Declarations earlier in this section for a complete list of recognized BPS rates.

Example

```
AddModemCode(M, 17, mrConnect, mnErrorControl+mn9600BPS);
```

This creates a new response in the numeric code response table for a modem response of 17. This response means that an error control connection was established at 9600 BPS. When using objects, the M parameter would be omitted.

See Also

`RemoveModemCode`

Syntax

```
procedure AddModemCommand(M : ModemRecPtr; S : String; Cmd : Word; Flags : Word);  
procedure AbstractModem.AddModemCommand(S : String; Cmd : Word; Flags : Word);
```

Purpose

Add or change a modem command in the modem command table.

Description

This routine can be used to add a new command or to change an existing one in the specified modem's command table. If there is not enough room in the table for the new/changed command, an `ecTableFull` error is generated.

`Cmd` is the Async Professional code for the modem command (`mcNone` through `mcUser9`). `S` is the string of characters to be sent to the modem in order to execute the command. Note that `mcNone` is reserved for internal use by `RemoveModemCommand`. Assigning a string to `mcNone` does nothing.

The `Flags` parameter is a bit-mapped word that indicates how the command string is to be constructed. If the `moUsePrefix` bit is set, the string actually sent to the modem is preceded by the 'AT' prefix. If the `moUseSuffix` bit is set, the command string is followed by a carriage return. If the `moDialTimeout` bit is set, the dial timeout value (default = 60 seconds) is used when waiting for a response from the modem to the command, rather than the shorter command timeout value (default = 10 seconds).

`AddModemCommand` has two principal uses: 1) to alter the string sent to the modem for a pre-defined modem command, in order to deal with a particular modem, and 2) to define a new command supported by the current modem.

Example

```
AddModemCommand(M, mcUser0, 'S0=1', moUsePrefix+moUseSuffix);
```

This creates a new command, `mcUser0`, that activates auto-answer mode by sending the string 'ATS0=1' to the modem, followed by a carriage return. The command can be executed using `ExecuteModemCommand(M, mcUser0, -1);`. When using objects, the `M` parameters would be omitted.

See Also

`ExecuteModemCommand`

`RemoveModemCommand`

AddModemResponse

Syntax

```
procedure AddModemResponse(M : ModemRecPtr; S : String; Code : Word);  
procedure AbstractModem.AddModemResponse(S : String; Code : Word);
```

Purpose

Insert a new response in the modem response table.

Description

This routine can be used to add a new response in the specified modem's response table. Code is the Async Professional code for the modem response (mrOK through mrNone). S is the string of characters generated by the modem to indicate a given response. If there is not room in the table for the new/changed response, an ecTableFull error is generated.

Unlike AddModemCommand, AddModemResponse never changes an existing entry in the table it works with. It always tries either to add to the end of the table or, if a response of the same length has been previously removed by calling RemoveModemResponse, it reuses its space.

Note that mrNone is reserved for internal use by RemoveModemResponse. Assigning a response code of mrNone to a string does nothing.

Example

```
AddModemResponse(M, 'NOANSWER', mrNoAnswer);
```

This associates the mrNoAnswer response code with the string 'NOANSWER' (a purely hypothetical example). When using objects, the M parameter would be omitted.

See Also

RemoveModemResponse

Syntax

```
procedure AnswerModem(M : ModemRecPtr);  
procedure AbstractModem.AnswerModem;
```

Purpose

Tell the modem to answer immediately.

Description

This routine executes the modem's answer command (normally 'ATA'), which tells the modem to immediately "pick up the phone" and try to respond to a calling modem.

Note that this routine does *not* place the modem in auto-answer mode. That is accomplished by setting S-register 0 to a non-zero value using SetModemRegister.

Example

```
WriteLn('Waiting for call...');  
Finished := False;  
repeat  
  {Check for ring}  
  if CheckDeltaRI(P) then begin  
    {Answer the phone}  
    AnswerModem(M);  
    case AsyncStatus of  
      ecConnect : WriteLn('Connect');  
      {... handle errors ...}  
    end;  
    Finished := True;  
  end;  
  if KeyPressed then  
    {check key -- see if user wants to halt} ;  
until Finished;  
WriteLn('Entering terminal mode');  
{...}
```

This example sits in a loop waiting for the phone to ring. When it does, AnswerModem is called to answer the phone. The program then goes into a terminal emulation mode.

When using objects, the P parameter to CheckDeltaRI and the M parameter to AnswerModem would be omitted.

See Also

AutoAnswerModem
CheckDeltaRI (APCOM)

AbstractPort.CheckDeltaRI
SetModemRegister

AutoAnswerModem

Syntax

```
procedure AutoAnswerModem(M : ModemRecPtr);  
procedure AbstractModem.AutoAnswerModem;
```

Purpose

Wait for an answer from a modem in auto-answer mode.

Description

This procedure can be used to detect and answer incoming calls when your modem is in auto-answer mode (i.e., when register S0 has any value other than zero).

This procedure assumes that the modem object's known values for registers S0 and S7 are correct. Make sure you call GetModemRegister for both of those registers (the modem object will save the register values internally).

AutoAnswerModem returns ecConnect if the modem detects and answers an incoming call. It returns ecUserAbort if your user abort function requests an abort. It returns ecNoCarrier if a connection could not be established (if, for example, the caller hung up before the modem saw the specified number of rings).

Example

```
WriteLn('Waiting for call...');  
{Answer the phone}  
AutoAnswerModem(M);  
case AsyncStatus of  
  ecConnect : WriteLn('Connect');  
  {... handle errors ...}  
end;  
WriteLn('Entering terminal mode');  
{...}
```

This example is functionally identical to the one for AnswerModem. It waits for the phone to ring. When it does, the program then goes into a terminal emulation mode.

When using objects, the P parameter to CheckDeltaRI and the M parameter to AnswerModem would be omitted.

See Also

AnswerModem

Syntax

```
procedure DialModem(M : ModemRecPtr; TelNo : NumberStr);  
procedure AbstractModem.DialModem(TelNo : NumberStr); virtual;
```

Purpose

Dial the specified telephone number.

Description

This routine tells the modem to dial the specified telephone number. The string should contain nothing except the phone number itself. DialModem automatically adds the string representing the modem's dial command (normally 'ATD') and any dial prefix previously specified by a call to SetDialPrefix. That is, the string sent by DialModem has the following form:

'ATD' + DialPrefix (if any) + TelNo + 'M'

Hayes-compatible modems ignore most embedded characters that are likely to be found in a phone number string. For example, the following are all considered equivalent:

```
1-415-555-1234  
1 415 555 1234  
1 (415) 555 1234  
14155551234
```

However, the following characters have special meaning when found within a phone number (they can be used either in an individual phone number passed to DialModem or in a dial prefix passed to SetDialPrefix):

- , insert a 2-second delay—e.g., '9,555-1234' dials 9, waits 2 seconds, then dials 555-1234
- @ wait for 5 seconds of silence before proceeding (not available on all modems)
- w wait for a dial tone before proceeding—e.g., '9w555-1234' dials 9, waits for a dial tone, then dials 555-1234 (not available on all modems)
- ; if used as the last character in a dialing command, the modem returns to command state immediately after dialing, and will not respond to a carrier signal on the other end of the phone line
- " tells the modem to treat alphabetic characters as numbers—e.g., the modem would interpret "'1-800-hot-cars' as '1-800-468-2277' (not available on all modems)
- r reverse originate/answer frequencies; used before or after the phone number when calling a modem that is designed only to originate phone calls, not to receive them
- ! flash the switch hook; the modem goes off hook for .5 seconds, on hook for .5 seconds, then off hook again; used with certain phone systems to transfer calls, etc., as well as in certain cases where multiple modems share a single phone line (not available on all modems)

Typically, your modem manual documents these special characters in greater detail, as well as any others that it recognizes.

Example

```
DialModem(M, '9,555-4321');
```

Dial 9, wait two seconds, then dial 555-4321. When using objects, the M parameter would be omitted.

See Also

GetConnectSpeed
GetLastErrorMode

SetDialPrefix

Done / DoneModem

Syntax

destructor AbstractModem.Done; virtual;

Purpose

Dispose of this object.

Description

This destructor disposes of the modem object and any dynamic memory previously allocated by it. It does nothing to the port associated with the modem.

Example

See the example for HayesModem.Init.

See Also

DoneModem

HayesModem.Init

Syntax

procedure DoneModem(M : ModemRecPtr);

Purpose

Dispose of this modem.

Description

This routine disposes of the specified modem record. It does nothing to the port associated with the modem record.

Example

See the example for InitModem.

See Also

AbstractModem.Done

InitModem

Syntax

```
procedure ExecuteModemCommand(M : ModemRecPtr; Cmd : Word; CmdVal : Integer);
procedure AbstractModem.ExecuteModemCommand(Cmd : Word;
                                             CmdVal : Integer); virtual;
```

Purpose

Execute a modem command.

Description

This procedure is used to send the specified modem the command string associated with Cmd. If no such command exists in M's command table, an ecNullCommand error is generated.

CmdVal is a numeric argument to be sent as part of the command string. For example, if the string to be sent is 'ATY0', and 'Y' is assigned to mcUser0, you would use ExecuteModemCommand(M, mcUser0, 0). If CmdVal is less than 0, no numeric argument is sent to the modem.

If the automatic response handling option is on, as it is by default, ExecuteModemCommand waits for the modem to respond before returning. Its response can be obtained by checking AsyncStatus. If response handling is not on, you must call GetModemResponse manually.

Examples

```
ExecuteModemCommand(M, mcPulse, -1);
```

Send the mcPulse command to the modem, with no numeric argument. This is the same as calling SetDialPulse. The resulting command string is normally 'ATP'. When using objects, the M parameter would be omitted.

```
ExecuteModemCommand(M, mcEcho, 1);
```

Send the mcEcho command to the modem, with a numeric argument (1) that tells it to turn echo on. This is the same as calling SetModemEcho with a final parameter of mEchoOn. The resulting command string is normally 'ATE1'.

See Also

PutModemCommand

SetHandleResponses

Syntax

```
function GetConnectSpeed(M : ModemRecPtr) : LongInt;
function AbstractModem.GetConnectSpeed : LongInt;
```

Purpose

Return the current connect speed of the modem.

Description

This routine can be called at any time after a call to DialModem (assuming that the HandleResponses option is on, as it is by default) to determine the connect speed reported by the modem. For example, if the modem returns 'CONNECT 2400' or the equivalent numeric code in response to the dial command, then GetConnectSpeed returns 2400. If the connect speed is unknown, or if DialModem has not yet been called, GetConnectSpeed returns 0.

Example

```
DialModem(M, '9,555-4321');
{assume response is 'CONNECT 1200'}
Speed := GetConnectSpeed(M);
```

GetConnectSpeed returns 1200. When using objects, the M parameters would be omitted.

See Also

DialModem

SetHandleResponses

GetLastCode / GetLastErrorMode

Syntax

```
function GetLastCode(M : ModemRecPtr) : Integer;  
function AbstractModem.GetLastCode : Integer;
```

Purpose

Return the code of the last modem response.

Description

If the modem is configured to return codes as responses, this function returns the modem's numeric response to the last modem command. If the modem is configured to return verbal responses (in which case `GetLastText` should be called instead), `GetLastCode` returns -1.

If the `HandleResponses` option was turned off by calling `SetHandleResponses`, it is necessary to call `GetModemResponse` first so that the response can be extracted from the stream of characters coming in from the modem.

Example

```
DialModem(M, '9,555-4321');  
{assume success and a connection speed of 1200}  
Code := GetLastCode(M);  
  
GetLastCode returns 5 (= 'CONNECT 1200'). When using objects, the M parameters would be omitted.
```

See Also

<code>GetLastText</code>	<code>SetHandleResponses</code>
<code>GetModemResponse</code>	<code>SetModemResults</code>

Syntax

```
function GetLastErrorMode(M : ModemRecPtr) : Boolean;  
function AbstractModem.GetLastErrorMode : Boolean;
```

Purpose

Return True if the last connection was an error correcting one.

Description

Modems using MNP or V.42 can provide error control (error detection and correction) when connected with other MNP or V.42 modems with the error control feature enabled.

When a connection is established via `DialModem` or `AnswerModem`, `GetLastErrorMode` can be used to determine whether the connection will use error control. When using verbal responses, the modem can be instructed *not* to specify whether the connection uses error control. If your modem is operating in this mode, then `GetLastErrorMode` will always return False.

Example

```
DialModem(M, '555-1212');  
if AsyncStatus = ecConnect then begin  
  WriteLn('Connection established');  
  WriteLn('BPS: ', GetLastConnectSpeed(M));  
  if GetLastErrorMode(M) then  
    WriteLn('Error control is provided')  
  else  
    WriteLn('No error control is provided');  
end;
```

If the call to `DialModem` results in a successful connection, the BPS rate and the error control mode are displayed. When using objects, the M parameters would be omitted.

See Also

<code>GetLastCode</code>	<code>GetLastText</code>
--------------------------	--------------------------

Syntax

```
function GetLastText(M : ModemRecPtr) : String;  
function AbstractModem.GetLastText : String;
```

Purpose

Return the text of the last verbal modem response.

Description

If the modem is configured to return words as responses, this function returns the string that was the modem's response to the last modem command. If the modem is configured to return code responses (in which case GetLastCode should be called instead), GetLastText returns an empty string.

If the HandleResponses option was turned off by calling SetHandleResponses, it is necessary to call GetModemResponse first so that the response can be extracted from the stream of text coming in from the modem.

Example

```
DialModem(M, '9,555-4321');  
{assume success and a connection speed of 1200}  
S := GetLastText(M);
```

GetLastCode returns 'CONNECT 1200'. When using objects, the M parameters would be omitted.

See Also

GetLastCode	SetHandleResponses
GetModemResponse	SetModemResults

Syntax

```
function GetModemRegister(M : ModemRecPtr; Reg : Integer) : Byte;  
function AbstractModem.GetModemRegister(Reg : Integer) : Byte;
```

Purpose

Return the value stored in the specified S-register.

Description

This function returns the value stored in the specified S-register (Reg) of the modem by sending it the 'ATS n ?' command, where n is the number of the S-register to be queried.

See the entry for SetModemRegister for a table that shows the uses of the standard S-registers.

Example

```
if GetModemRegister(M, 0) <> 0 then  
  WriteLn('Auto-answer is on');
```

Checks to see if the modem is configured for auto-answer. When using objects, the M parameter would be omitted.

See Also

SetModemRegister

GetModemResponse

Syntax

```
procedure GetModemResponse(M : ModemRecPtr; CurTimeout : Integer);  
procedure AbstractModem.GetModemResponse(CurTimeout : Integer); virtual;
```

Purpose

Wait for the result from the last modem command.

Description

This procedure should be called when the automatic response handling option is off (it's on by default) to obtain the result reported by the modem to the last command sent to it.

That result is returned in AsyncStatus, which is typically one of the following: ecOK, ecConnect, ecNoCarrier, ecError, ecNoDialtone, ecBusy, ecNoAnswer, ecVoice, or ecRing. If the operation timed out while waiting for a response, an ecTimeout error is reported. If the modem's response cannot be understood, the error is ecUnknownModemResult.

CurTimeout indicates the amount of time to wait (in clock ticks) for the modem's response. The recommended values are DefTimeout (182, or 10 seconds) for most modem commands, and DefDialTimeout (1092, or 60 seconds) for dial and answer commands.

After a call to GetModemResponse, the actual response string returned by the modem can be obtained by calling GetLastText. If numeric responses were selected, call GetLastCode instead.

If you turned automatic response handling off, by calling SetHandleResponses(False), then you need to call GetModemResponse yourself.

If you call GetModemResponse when automatic response handling is *on* (which is an error) GetModemResponse waits DefTimeout clock ticks for another response and return ecTimeout if none is received.

Example

```
DialModem(M, '555-1234');  
GetModemResponse(M, DefDialTimeout);
```

Dial a phone number and wait for the modem's response. When using objects, the M parameters would be omitted.

See Also

GetLastCode
GetLastText
SetHandleResponses

SetModemResults
SetModemTimeouts

Syntax

```
procedure HangupModem(M : ModemRecPtr; Opt : Word; DropDTR : Boolean);  
procedure AbstractModem.HangupModem(Opt : Word; DropDTR : Boolean);
```

Purpose

Hangup the modem by sending the hangup string or dropping DTR.

Description

This routine hangs up the modem in one of two ways. If DropDTR is True, it does so by dropping the DTR line for 8 ticks (about 1/2 second) and then raising it again. The duration is controlled by global variable DTRDropHold.

If DropDTR is False, it sends the modem the mcHook command (usually 'ATH') along with the indicated argument (Opt). If Opt is mOnHook, the phone is hung up. If Opt is mOffHook, the phone remains on line. If the phone is currently "on hook" and you pass mOffHook, the modem takes the phone "off hook" (the opposite of hanging up). While there's little reason to ever do this, the modem command 'ATH' allows it and, therefore, so does HangupModem.

Since dropping DTR is a far more reliable method of hanging up the phone than the "ATH" command, it is recommended that you always pass True for DropDTR.

Examples

```
HangupModem(M, -1, True);
```

Hang up the modem by dropping DTR. When using objects, the M parameter would be omitted.

```
HangupModem(M, mOnHook, False);
```

Hang up the modem by sending it the mcHook command, and tell it to hang up the phone.

See Also

SetDTR (APCOM)

AbstractPort.SetDTR

Syntax

```
constructor HayesModem.Init(AP : AbstractPortPtr);
```

Purpose

Allocate and initialize a modem object.

Description

This constructor initializes a modem object used to communicate with a Hayes-compatible modem attached to the com port associated with AP^.

This constructor is used by both CourierModem and MicrocomModem, which are derived from HayesModem. Although the AbstractModem object defines an Init constructor (called by HayesModem) that initializes most of the fields in the object, you would not call it yourself unless you were deriving your own modem object from AbstractModem.

Example

```
HM.Init(@UP);
```

```
...
```

```
HM.Done;
```

Initializes HM (HayesModem) with a port named UP; later destroys modem object.

See Also

Done

InitModem

InitModem

Syntax

```
procedure InitModem(var M : ModemRecPtr; P : PortRecPtr);
```

Purpose

Allocate and initialize a modem.

Description

This routine is the procedural equivalent of HayesModem.Init. It initializes a modem record that is used by the various routines in APMODEM to communicate with a Hayes-compatible modem attached to the com port associated with P^.

Note that, unlike OOMODEM, APMODEM does not provide any specific support for Courier or Microcom modems. To take advantage of their special features, you must use the APCOUR and APMCOM units, documented in §6.D and §6.E.

Example

```
InitPortFast(P, Com1, 2400);  
InitModem(M, P);  
...  
DoneModem(M);  
DonePort(P);
```

Initialize P for Com1 and modem record pointer M; later dispose of both P and M.

See Also

DoneModem

HayesModem.Init

Syntax

```
constructor AbstractModem.Load(var S : IdStream);
```

Purpose

Load a modem object from a stream.

Description

This constructor loads a modem object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Store.

The stream registration routines for the modem object types are AbstractModemStream, HayesModemStream, CourierModemStream, MicrocomModemStream, NullModemStream, and AllModemStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses
  OpRoot, ApPort, OoCom, OoModem;
var
  UP : UartPort;
  HM : HayesModem;
  S : BufIdStream;
  Status : Word;
begin
  if not S.Init('MODEM.STM', SOpen, 1024) then begin
    WriteLn('Error opening stream');
    Halt;
  end;

  S.RegisterHier(AllModemStream);
  S.RegisterHier(UartPortStream);
  S.Get(UP);
  S.Get(HM);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Load error: ', Status);
    Halt;
  end;

  WriteLn('Load successful');
  S.Done;
  UP.Done;
  HM.Done;
end.
```

Instantiates UP and HM from a stream created by the Store example using the stream's Get method, which calls the objects' Load constructors indirectly.

PutModemCommand

Syntax

```
procedure PutModemCommand(M : ModemRecPtr; Cmd : String);  
procedure AbstractModem.PutModemCommand(Cmd : String); virtual;
```

Purpose

Send a command string directly to the modem.

Description

This procedure does the same thing as ExecuteModemCommand (which calls it), except that it bypasses the Async Professional system of modem command codes and allows you to send a string (Cmd) directly to the modem.

Note that it is your responsibility to supply a complete command string, including the AT prefix and a carriage return at the end of the string if appropriate.

Example

```
PutModemCommand(M, 'ATDT555-1234' ^ M);
```

This command dials 555-1234 on a touch-tone phone. When using objects, the M parameter would be omitted.

See Also

ExecuteModemCommand

RemoveModemCode / RemoveModemCommand

Syntax

```
procedure RemoveModemCode(M : ModemRecPtr; NC : Byte);  
procedure AbstractModem.RemoveModemCode(NC : Byte);
```

Purpose

Remove a modem numeric response from the response table.

Description

This routine removes an entry from the modem's numeric code response table. NC is the numeric code to be removed. If the code is not found in the table, an ecNullCommand error is generated.

Example

```
AddModemCode(M, 17, mrConnect, mnErrorControl+mn9600BPS);  
...  
RemoveModemCode(M, 17);
```

Adds a response code of 17 to the modem's code response table and later removes it. When using objects, the M parameters would be omitted.

See Also

AddModemCode	RemoveModemResponse
AddModemResponse	

Syntax

```
procedure RemoveModemCommand(M : ModemRecPtr; Cmd : Word);  
procedure AbstractModem.RemoveModemCommand(Cmd : Word);
```

Purpose

Remove a modem command from the modem command table.

Description

This routine removes an entry from the modem's command table. Cmd is the code for the command to be removed. If there are multiple command strings assigned to the same code, RemoveModemCommand removes only the first one that it finds. If there is no command assigned to the specified code, an ecNullCommand error is generated.

Example

```
AddModemCommand(M, mcUser0, 'S0=1', moUsePrefix+moUseSuffix);  
...  
RemoveModemCommand(M, mcUser0);
```

Adds a user-defined command to the modem's command table; later removes it. When using objects, the M parameters would be omitted.

See Also

AddModemCommand	RemoveModemResponse
-----------------	---------------------

RemoveModemResponse / RepeatModemCommand

Syntax

```
procedure RemoveModemResponse(M : ModemRecPtr; S : String);  
procedure AbstractModem.RemoveModemResponse(S : String);
```

Purpose

Remove a modem response from the modem response table.

Description

This routine removes an entry from the modem's response table. S is the response string to be removed. If the string is not found in the response table, an ecNullCommand error is generated.

Example

```
AddModemResponse(M, 'NOANSWER', mrNoAnswer);  
...  
RemoveModemResponse(M, 'NOANSWER');
```

Adds 'NOANSWER' to the modem's response table; later removes it. When using objects, the M parameters would be omitted.

See Also

AddModemResponse

RemoveModemCommand

Syntax

```
procedure RepeatModemCommand(M : ModemRecPtr);  
procedure AbstractModem.RepeatModemCommand;
```

Purpose

Repeat the last modem command.

Description

This routine sends the modem the mcRepeat command (usually 'A'), which tells it to repeat the last modem command. The most common use for this command is to redial a phone number that was busy on the previous attempt.

Example

```
I := 1;  
DialModem(M, '555-1234');  
while (I < 20) and (AsyncError <> ecConnect) do begin  
  Inc(I);  
  Delay(20000);  
  RepeatModemCommand(M);  
end;
```

Try up to 20 times to dial 555-1234, pausing 20 seconds between retries. When using objects, the M parameters would be omitted.

See Also

DialModem

Syntax

```
procedure ResetModem(M : ModemRecPtr);  
procedure AbstractModem.ResetModem;
```

Purpose

Reset the modem to the power-on defaults.

Description

This procedure returns the modem to the state it was in when it was turned on by issuing the command 'ATZ'. Although this routine is provided primarily for the sake of completeness, it can be useful in certain cases. For example, if you're using a high-end modem with non-volatile memory, you can configure the modem with desired settings once, write them to non-volatile memory (on a Courier modem, the command to do so is '&W'), then restore them at the beginning of a program by calling ResetModem.

Example

```
ResetModem(M);
```

Reset the modem to its default state. When using objects, the M parameter would be omitted.

Syntax

```
procedure SetCarrierTrans(M : ModemRecPtr; Opt : Word);  
procedure AbstractModem.SetCarrierTrans(Opt : Word);
```

Purpose

Turn carrier transmitter on/off.

Description

This procedure can be used to tell the modem to turn its internal carrier signal on (Opt = mTransOn, command = 'ATC1') or off (Opt = mTransOff, command = 'ATC0').

The default state of On is correct for virtually all situations. You would typically turn the carrier signal off only if two terminals/modems were connected to the same phone line, with the second being used to monitor the data being sent to the first. By turning off the carrier signal for the second modem, it can operate in a receive-only mode and not interfere with the conversation between the first modem and the remote.

Example

```
SetCarrierTrans(M, mTransOn);
```

Turn carrier signal on. When using objects, the M parameter would be omitted.

SetDataCompression / SetDCDControl

Syntax

```
procedure AbstractModem.SetDataCompression(State : Boolean); virtual;
```

Purpose

Turn data compression on or off.

Description

This routine turns data compression on or off for modems that have this facility built in, such as the Courier and Microcom modems. If this feature is enabled, the modem negotiates with the remote modem when making a connection to determine whether or not it implements a compatible form of data compression. If it is disabled, or if the remote does not provide data compression, data is sent/received normally. Modem manufacturers generally recommend that you turn data compression off when transmitting data that is already compressed (ARC, PAK, ZIP, or LZH files, for example).

This feature is available only on modems that support "managed modem links" such as MNP or V.42. If this feature is not supported by the modem object in use (HayesModem, for example), an `ecNullCommand` error is generated.

Example

```
CourierM.SetDataCompression(False);  
{... transfer a compressed file ...}  
CourierM.SetDataCompression(True);
```

Turn data compression off, transfer a compressed file, then turn it back on.

See Also

`SetDataCompression (APCOUR)`

`SetDataCompression (APMCOM)`

Syntax

```
procedure SetDCDControl(M : ModemRecPtr; Opt : Integer);  
procedure AbstractModem.SetDCDControl(Opt : Integer);
```

Purpose

Set Data Carrier Detect behavior.

Description

This procedure tells the modem how to set the DCD status line (data carrier detect). The two options are `mDCDAlwaysOn` and `mDCDFollowConnect`.

Specify `mDCDAlwaysOn` if you want the modem to turn on DCD and leave it on at all times. Specify `mDCDFollowConnect` if you want the modem to turn on DCD *only* when it has a connection with a remote modem. This is generally the preferred behavior. It allows your program to use `CheckDCD` to determine whether it has a connection or not. More specifically, you can use `CheckDCD` to check whether the remote modem dropped the connection.

Although this command is part of the Hayes command table, not all modems support it.

Example

```
SetDCDControl(M, mDCDFollowConnect);
```

Tells the modem to turn on the DCD signal *only* when it is connected to another modem. When using objects, the `M` parameter would be omitted.

See Also

`CheckDCD (APCOM)`
`AbstractPort.CheckDCD`

`SetDTRControl`

Syntax

```
procedure SetDialPrefix(M : ModemRecPtr; Prefix : PrefixStr);  
procedure AbstractModem.SetDialPrefix(Prefix : PrefixStr);
```

Purpose

Set a dialing prefix (do *not* include 'ATD').

Description

This routine allows you to specify a string of characters (Prefix) that are sent to the modem by DialModem between the dial command and the phone number. See the entry for DialModem for further details, including a list of special characters/symbols that might be found in a dialing prefix.

Example

```
SetDialPrefix(M, '9,');  
...  
DialModem(M, '555-4321');  
...  
DialModem(M, '555-1234');
```

Equivalent to dialing '9,555-4321' and later '9,555-1234'. When using objects, the M parameters would be omitted.

See Also

DialModem

Syntax

```
procedure SetDialPulse(M : ModemRecPtr);  
procedure AbstractModem.SetDialPulse;
```

Purpose

Set the dial mode to pulse.

Description

This procedure configures the modem to dial in pulse (rotary) mode, which is usually the default setting.

Example

```
if UsePulseToDial then  
  SetDialPulse(M)  
else  
  SetDialTone(M);
```

Selects pulse or tone dialing based on the value of UsePulseToDial. When using objects, the M parameters would be omitted.

See Also

SetDialTone

SetDialTone / SetDTRControl

Syntax

```
procedure SetDialTone(M : ModemRecPtr);  
procedure AbstractModem.SetDialTone;
```

Purpose

Set the dial mode to tone.

Description

This procedure configures the modem to use tone dialing, which is significantly faster than pulse dialing, and should be used if the phone line allows it.

Example

See the example for SetDialPulse.

See Also

SetDialPulse

Syntax

```
procedure SetDTRControl(M : ModemRecPtr; Opt : Integer);  
procedure AbstractModem.SetDTRControl(Opt : Integer);
```

Purpose

Set Data Terminal Ready behavior.

Description

This procedure can be used to tell the modem how to interpret the DTR signal from your PC. The two options are mDTRAlwaysOn and mDTRTerminateCall.

Specify mDTRAlwaysOn if you want the modem to assume that the DTR signal is always on; that is, if you want it to ignore whatever the PC happens to do with the DTR signal.

Specify mDTRTerminateCall if you want the modem to monitor the DTR signal and terminate the current connection should the PC lower the DTR signal. This is generally the preferred option. It gives your program the ability to disconnect the call at any time, by calling SetDTR(False). It can also be used to terminate the call if your program halts unexpectedly, before it has a chance to clean up (assuming that you specified the ptDropModemOnClose option, which is the default).

Although this command is part of the Hayes command table, not all modems support it.

Example

```
SetDTRControl(M, mDTRTerminateCall);
```

Tells the modem to hang up if the DTR signal is lowered. When using objects, the M parameter would be omitted.

See Also

SetDCDControl
SetDTR (APCOM)

AbstractPort.SetDTR

Syntax

```
procedure AbstractModem.SetErrorControl(ErrorOpt : ErrorStates); virtual;
```

Purpose

Turn error control on or off.

Description

This routine can be used to configure the modem to use its built-in error correction logic when communicating with a remote modem that provides the same facility. The possible values for ErrorOpt are eCheckOff (error checking off), eCheckOn (error checking on), and eCheckAuto (error checking used if possible).

If the option selected is eCheckAuto, the modem negotiates with the remote modem when making a connection to determine whether or not it implements error correction. If it doesn't, error correction is not used. If eCheckOn is selected and the remote modem does not provide error correction, the local modem breaks the connection.

This feature is available only on modems that support "managed modem links" such as MNP or V.42. If this feature is not supported by the modem object in use (HayesModem, for example), an ecNullCommand error is generated.

Example

```
CourierM.SetErrorControl(eCheckAuto);  
CourierM.DialPhone('555-1234');
```

Enable error correction and call 555-1234. If the answering modem supports a compatible error control protocol and has that feature turned on, the two modems will employ error control when transferring data.

See Also

SetErrorControl (APCOUR)

SetErrorControl (APMCOM)

SetFlowControl

Syntax

```
procedure AbstractModem.SetFlowControl(FlowOpts : Word); virtual;
```

Purpose

Set flow control options.

Description

This routine enables flow control between the modem and the terminal. This feature is necessary (and possible) *only* with "managed modem links" using MNP or V.42. If your modem doesn't have this feature or you are not using it, then you would just use the software flow control facilities of APCOM/OOCOM.

FlowOpts is a bit-mapped word that describes the types of flow control desired, using one or more of the following option flags:

The fSWTrans option tells the modem to send an Xoff character to the terminal when its transmit buffer fills to stop the terminal from transmitting characters, and an Xon character to tell it to resume.

The fSWRec option tells the modem that the terminal will use Xon and Xoff characters to control the flow of data received by the terminal from the modem. These characters are not sent to the remote.

The fHWTrans option serves the same purpose as fSWTrans, except that it raises/lowers the CTS (Clear To Send) signal on the terminal's UART rather than using Xon/Xoff.

The fHWRec option serves the same purpose as fSWRec, except that it raises/lowers the RTS (Request To Send) signal on the terminal's UART rather than using Xon/Xoff.

The fRmtXoff option works in conjunction with the fSWRec and fSWTrans options. It tells the modem not to act on received Xon/Xoff characters but to send them to the remote modem instead. Typically, you would use this mode only if you are using an MNP or V.42 modem but are *not* using its error control features (consult your modem manual for more information).

Keep in mind that if you enable hardware or software flow control on the modem's side, you must also enable it at the terminal's side, by calling HWFlowEnable or SWFlowEnable.

If this feature is not supported by the modem object in use (HayesModem, for example), an ecNullCommand error is generated.

Examples

```
CourierM.SetFlowControl(fHWTrans+fHWRec);
```

Enable hardware flow control in both directions.

```
CourierM.SetFlowControl(fSWTrans+fSWRec);
```

Enable software flow control in both directions.

See Also

HWFlowEnable (APCOM)
AbstractPort.HWFlowEnable
SetFlowControl (APCOUR)

SetFlowControl (APMCOM)
SWFlowEnable (APCOM)
AbstractPort.SWFlowEnable

Syntax

```
procedure SetHandleResponses(M : ModemRecPtr; State : Boolean);  
procedure AbstractModem.SetHandleResponses(State : Boolean);
```

Purpose

Turn automatic response handling on or off.

Description

This routine turns automatic response handling on (State = True) or off (State = False). If this option is on, as it is by default, all routines that execute modem commands (including the low-level routines PutModemCommand and ExecuteModemCommand) automatically call GetModemResponse to obtain the modem's response before they return.

Example

```
SetHandleResponses(M, False);  
...  
DialModem(M, '555-1234');  
GetModemResponse(M, DefDialTimeout);
```

Turn automatic response handling off, dial the phone, and wait for a response. When using objects, the M parameters would be omitted.

See Also

ExecuteModemCommand
GetModemResponse

PutModemCommand

SetLinkLocked

Syntax

procedure AbstractModem.SetLinkLocked(Locked : Boolean); virtual;

Purpose

Lock the link at the current rate or follow the connection rate.

Description

This routine tells the modem how to adjust the speed between the modem and the PC (the DCE and DTE)—whether to lock the DTE/DCE link at the current rate or have the modem change the rate to follow the connection rate.

In unmanaged modem links (non-MNP and non-V.42) you would *not* want to lock the DTE/DCE link; instead you would want the DTE/DCE link to follow the connection speed (since there's no reason for doing otherwise).

In managed modem links (MNP or V.42) you quite likely *do* want to lock the DTE/DCE rate. That is, through synchronization techniques and data compression, such links can generally transfer data faster than an equivalent unmanaged link. For example, an MNP connection can theoretically transfer data at 276 characters per second over a 2400 BPS link, much higher than the unmanaged connection speed of 240 characters per second. If the DTE/DCE link followed the connection speed of 2400 BPS, then you wouldn't see any of the benefit of the improved throughput (since the DTE is only providing data at 240 characters per second). However, if you lock the DTE/DCE link at 4800 BPS then the modem can achieve the higher throughput. Of course, you would also need to enable flow control since the DTE and DCE are now operating at different speeds.

This feature is available only on modems that support "managed modem links" such as MNP or V.42. If this feature is not supported by the modem object in use (HayesModem, for example), an `ecNullCommand` error is generated.

Example

```
New(ComPort, InitCustom(Com1, 19200, NoParity, 8, 1, 2000, 2000, MyOpts));
ComPort^.HWFlowEnable(1800, 200, hfUseRTS + hfRequireCTS);
New(Modem, ComPort);
Modem^.SetFlowControl(fHWTrans + fHWRec);
Modem^.SetLinkLocked(True);
```

The `ComPort` object opens `Com1` at 19200 baud with input/output buffers of 2000 bytes. It also requests complete transmit and receive hardware flow control using RTS/CTS. The `Modem` object also requests transmit and receive hardware flow control. Finally, the DTE/DCE link is locked at 19200 (its current speed). Now, the DTE/DCE link will continue to operate at 19200 no matter what the actual connection speed is between the local and remote modems.

See Also

`SetLinkLocked (APCOUR)`
`SetLinkLocked (APMCOM)`

`SetModemSpeed`

Syntax

```
procedure SetModemCmdMode(M : ModemRecPtr);  
procedure AbstractModem.SetModemCmdMode;
```

Purpose

Switch the modem from terminal mode to command mode.

Description

This routine switches the modem from terminal (online) mode to command mode. This is done by sending the escape code (usually '+++') to the modem. Most modems require a one second delay before the escape code and another one afterwards. SetModemCmdMode provides these delays.

Example

```
SetModemCmdMode(M);  
{...send some modem commands...}  
SetModemOnline(M);
```

Switches the modem from terminal mode to command mode, issues additional modem commands, and returns the modem to terminal mode.

See Also

SetModemOnline

Syntax

```
procedure SetModemCmdTable(M : ModemRecPtr; P : Pointer; Max : Word);  
procedure AbstractModem.SetModemCmdTable(P : Pointer; Max : Word);
```

Purpose

Change the modem to use the specified command table.

Description

This routine changes the default command table assigned when M (or the modem object) was initialized. The most common use is when a modem you are using is radically different from any of the supported modem types. P is a pointer to the new command table. Max is the last index entry in the array, which is assumed to be zero-based.

Example

```
const  
  MyCommandMax = 110;  
  MyCommandSet : array[0..MyCommandMax] of Byte = (  
    {len flags chars          command type      modem command}  
    3, $07, Ord('A'),          mcAnswer,         {Answer phone}  
    4, $00, Ord('A'),Ord('/'), mcRepeat,         {Rpt cmd}  
    ...  
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0); {110}  
    ...  
  New(Modem, Init(ComPort));  
  Modem.SetModemCmdTable(@MyCommandSet, MyCommandMax);
```

MyCommandSet is a custom command table for a modem type that is not currently supported. After instantiating Modem, SetModemCmdTable is called to replace the original command table with MyCommandSet.

SetModemCodeSet

Syntax

```
procedure SetModemCodeSet(M : ModemRecPtr; Opt : Word);  
procedure AbstractModem.SetModemCodeSet(Opt : Word);
```

Purpose

Set the modem response code set.

Description

This routine configures the modem to use different response code sets to take advantage of call progress detection and other advanced features of the modem. To select code set 0, use Opt = mExtSet0; to select code set 1, use Opt = mExtSet1; etc.

All Hayes-compatible modems implement set 0, the set provided by the original Hayes Smartmodem. Most modems that support multiple sets use set 1 by default. Those that don't support multiple sets ignore this command and generate an ecNullCommand error.

The following table, taken from the U.S. Robotics Courier manual, shows the response codes that are possible for a given set. This table is *not* valid for all modems; it's shown as an example only. Consult your modem manual to see how code sets are implemented on your modem.

Result codes	0	1	2	3	4	5	6	7
0/OK	x	x	x	x	x	x	x	x
1/CONNECT	x	x	x	x	x	x	x	x
2/RING	x	x	x	x	x	x	x	x
3/NO CARRIER	x	x	x	x	x	x	x	x
4/ERROR	x	x	x	x	x	x	x	x
5/CONNECT 1200		x	x	x	x	x	x	x
6/NO DIAL TONE			x		x		x	x
7/BUSY				x	x	x	x	x
8/NO ANSWER				x	x	x	x	x
9/reserved				x	x	x	x	x
10/CONNECT 2400		x	x	x	x	x	x	x
11/RINGING						x	x	x
12/VOICE						x	x	
13/CONNECT 9600		x	x	x	x	x	x	x
18/CONNECT 4800		x	x	x	x	x	x	x

Example

```
SetModemCodeSet(M, mExtSet6);
```

Select response code set 6. When using objects, the M parameter would be omitted.

SetModemEcho / SetModemErrorString

Syntax

```
procedure SetModemEcho(M : ModemRecPtr; Opt : Word);  
procedure AbstractModem.SetModemEcho(Opt : Word);
```

Purpose

Turn modem echoing on or off.

Description

This procedure allows you to configure the modem to echo back the commands that it receives (Opt = mEchoOn, command = 'ATE1'), or to suppress the echoing of commands (Opt = mEchoOff, command = 'ATE0').

Normally this setting doesn't matter, since Async Professional handles modem responses for you and ignores echoed commands found in response strings. Used in conjunction with GetModemResponse, echoing may be useful for debugging purposes, however. Most Hayes-compatible modems default to echo on.

Example

```
SetModemEcho(M, mEchoOff);
```

Turn command echoing off. When using objects, the M parameter would be omitted.

See Also

SetModemOnlineEcho

Syntax

```
procedure SetModemErrorString(M : ModemRecPtr; EC : ModemErrorString);  
procedure AbstractModem.SetModemErrorString(EC : ModemErrorString);
```

Purpose

Change the modem error string.

Description

This routine is used to set the string that GetModemResponse (and, therefore, DialModem and AnswerModem) should look for to determine if a just-established connection is using error control. Note that this feature is available only on modems that support MNP or V.42.

This is handled automatically for the built-in CourierModem and MicrocomModem types. You need to call this only if you are using another modem type.

GetModemResponse assumes that the error correction string follows the connect message and a separator character, like so:

```
CONNECT 9600/ARQ
```

If your modem uses a separator character other than the slash, change the default setting by changing M^.SepChar (for example, M^.SepChar := ' '; if the separator character is a space).

Example

```
SetModemErrorString(M, 'FAST');
```

Prepares GetModemResponse for a modem that returns the string 'FAST' whenever it establishes a connection using error control.

Syntax

```
procedure SetModemOnline(M : ModemRecPtr);  
procedure AbstractModem.SetModemOnline;
```

Purpose

Switch the modem from command mode to terminal mode.

Description

This routine allows you to switch the modem from command mode to terminal mode. Generally, you would need to use this only after calling SetModemCmdMode.

Example

```
SetModemCmdMode(M);  
{...send some modem commands...}  
SetModemOnline(M);
```

Switches the modem from terminal mode to command mode, issues additional modem commands and returns the modem to terminal mode. When using objects, the M parameters would be omitted.

See Also

SetModemCmdMode

Syntax

```
procedure SetModemOnlineEcho(M : ModemRecPtr; Opt : Word);  
procedure AbstractModem.SetModemOnlineEcho(Opt : Word);
```

Purpose

Turn modem remote echoing on or off.

Description

This routine allows you to configure the modem to echo the characters it sends to the remote (Opt = mEchoOn, command = 'ATF1') or not to echo them (Opt = mEchoOff, command = 'ATF0'). The usual default for a Hayes-compatible modem is echo off, sometimes referred to as "half duplex" mode. The echo on state is often referred to as "full duplex" mode.

This command is generally used when implementing a terminal emulator that is communicating with a remote that does not itself echo back the characters that are sent to it (most BBS's and online services do). Since most terminal emulators implement "echoing" themselves, by displaying all characters entered by the user before sending them to the remote, the online echo feature is rarely used.

Example

```
SetModemOnlineEcho(M, mEchoOn);
```

Turn online modem echoing on. When using objects, the M parameter would be omitted.

See Also

SetModemEcho

SetModemPort / SetModemQuiet

Syntax

```
procedure SetModemPort(M : ModemRecPtr; P : PortRecPtr);  
procedure AbstractModem.SetModemPort(AP : AbstractPortPtr);
```

Purpose

Change the modem to use the specified port.

Description

This routine tells a modem object to use a different port object. It is especially important to call this method if the port object originally passed to the modem object's constructor has been deallocated and then reallocated. You cannot assume that the object will be allocated at the same place on the heap the second time.

Example

```
UPP1 := New(UartPortPtr, InitFast(Com1, 2400));  
UPP2 := New(UartPortPtr, InitFast(Com2, 2400));  
HM.Init(UPP1);  
...  
HM.SetModemPort(UPP2);
```

Initialize HM for use with Com1; later switch to Com2.

See Also

HayesModem.Init

Syntax

```
procedure SetModemQuiet(M : ModemRecPtr; Opt : Word);  
procedure AbstractModem.SetModemQuiet(Opt : Word);
```

Purpose

Set modem response mode (quiet/normal).

Description

This routine switches the modem from normal mode (Opt = mNormalResp, command = 'ATQ0') to quiet mode (Opt = mQuietResp, command = 'ATQ1'). In quiet mode, the modem does not respond after it receives a command.

If you enable quiet mode, you should be sure to disable automatic response handling by calling SetHandleResponses, to prevent Async Professional from waiting for modem responses that will never come.

This function is generally not very useful. It is provided primarily for the sake of completeness.

Example

```
SetHandleResponses(M, False);  
SetModemQuiet(M, mQuietResp);
```

Switch the modem to quiet mode. When using objects, the M parameters would be omitted.

See Also

SetHandleResponses

Syntax

```
procedure SetModemRegister(M : ModemRecPtr; Reg, Value : Integer);
procedure AbstractModem.SetModemRegister(Reg, Value : Integer);
```

Purpose

Set the specified modem's S-register.

Description

This routine changes the value of one of the modem's S-registers. Reg is the number of the S-register to set. Value is the value to be placed in the register.

If Reg is in the range 0 through MaxSRegs (12) and Value is -1, then SetModemRegister uses the value from the SRegsInit table. Otherwise, SetModemRegister sets the modem register to Value.

If Reg is in the range 0 through MaxSRegs and Value is outside the limits specified by the SRegsInit table, ecOutOfRange is returned. No range checking is done if Regs is not in the range 0 through MaxSRegs.

The following table briefly summarizes the uses of the standard S-registers and shows their *usual* default values. Register 0's default value is generally determined by a dip switch setting on the modem:

Reg	Def	Meaning
0	--	Number of rings on which to answer phone (0 disables auto-answer)
1	0	Number of rings received on an incoming call
2	43	ASCII decimal code for escape character (43 = '+')
3	13	Carriage return character (13 = '^M')
4	10	Line feed character (10 = '^J')
5	8	Backspace character (8 = '^H')
6	2	Number of seconds to wait before dialing
7	30	Number of seconds to wait for carrier (sometimes 60)
8	2	Number of seconds to pause for a comma (',') in a dial command
9	6	Carrier detect response time (in tenths of a second)
10	7	Carrier loss response time (in tenths of a second)
11	70	Time in milliseconds between touch tones when dialing
12	50	Duration of guard time for escape codes (in 50ths of a second)

For further details concerning the uses of the S-registers, consult the manual for your modem.

Example

```
SetModemRegister(M, 0, 1);
```

Enables auto-answer and tells the modem to answer the phone on the first ring. When using objects, the M parameter would be omitted.

See Also

GetModemRegister

SetModemRespTable

Syntax

```

procedure SetModemRespTable(M : ModemRecPtr; P : Pointer; Max : Word);
procedure AbstractModem.SetModemRespTable(P : Pointer; Max : Word);

```

Purpose

Change the modem to use a different word response table.

Description

This routine changes the default word response table assigned when M (or the modem object) was initialized. The most common use for this is when the modem you're using is radically different from any of the supported modem types.

P is a pointer to the new word response table. Max is the index of the last byte in the array, which is zero-based.

Example

```
const
  {Modem response text set}
  MyResponseMax = 120;
  MyResponseID : String[16] = 'my responses';
  MyResponseSet : array[0..MyResponseMax] of Byte = (
    {len  text fragment                                response code      meaning}
    8,   cC,cO,cN,cN,cE,cC,cT,                          mrConnect,      {Connect}
    11,  cN,cO,c_,cC,cA,cR,cR,cI,cE,cR,                  mrNoCarrier,    {No carrier}
    6,   cE,cR,cR,cO,cR,                                  mrError,        {Error}
    ...
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0); {120}
...
New(Modem, Init(ComPort));
Modem^.SetModemRespTable(@MyResponseSet, MyResponseMax);
...ResponseSet;
```

MyResponseSet is a custom response table for a modem type that is not currently supported. After instantiating Modem as a HayesModem, SetModemRespTable is called to replace the original command table with MyResponseSet.

Syntax

```
procedure SetModemResults(M : ModemRecPtr; Opt : Word);  
procedure AbstractModem.SetModemResults(Opt : Word);
```

Purpose

Set results to words or codes.

Description

This routine lets you configure the modem to generate responses either as numeric codes (Opt = mCodeResults, command = 'ATV0') or words (Opt = mWordResults, command = 'ATV1').

The default setting is word results. Since Async Professional is designed to work with either words (the default) or codes, this setting doesn't matter in most cases.

Example

```
SetModemResults(M, mCodeResults);
```

Tell the modem to respond using numeric codes. When using objects, the M parameter would be omitted.

See Also

GetLastCode

GetLastText

Syntax

```
procedure SetModemSpeaker(M : ModemRecPtr; Opt : Word);  
procedure AbstractModem.SetModemSpeaker(Opt : Word);
```

Purpose

Set the speaker mode.

Description

This command allows you to tell the modem when to turn on its internal speaker. Opt should be one of the following: mSpeakerOff (command = 'ATM0') turns the speaker off completely; mSpeakerOnDial (command = 'ATM1', usually the default setting) turns the speaker on when dialing and off when a carrier is detected; mSpeakerOn (command = 'ATM2') turns the speaker on and leaves it on; mSpeakerOnConnect (command = 'ATM3') turns the speaker on when the last digit of a phone number is dialed and off when a carrier is detected.

This function is available on all Hayes-compatible modems, with the exception of the mSpeakerOnConnect option, which is not.

Example

```
SetModemSpeaker(M, mSpeakerOff);
```

Turn the modem's speaker off completely. When using objects, the M parameter would be omitted.

See Also

SetModemVolume

SetModemSpeed / SetModemTimeouts

Syntax

procedure AbstractModem.SetModemSpeed(BPS : LongInt); virtual;

Purpose

Set the DCE/DCE data rate.

Description

This routine sets a specific phone link speed between two modems. If the speed you request can't be attained, the modem will not establish a connection.

This isn't a normal mode of operation; SetSpeedMatching is generally preferred because the modem accepts whatever speed the answering modem is using.

However, you may want to use SetModemSpeed if a low-speed connection is unacceptable. For example, if you need to transfer many megabytes of data in an automated overnight transfer, a low BPS rate might not leave you enough time to complete the transfer. In this case, you would want your program to reject all low speed connections and retry for the desired speed.

This feature is available only on modems that support "managed modem links" such as MNP or V.42. If this feature is not supported by the modem object in use (HayesModem, for example), an ecNullCommand error is generated.

Example

```
SetModemSpeed(M, 9600);
```

After this call, the modem connects with a remote modem only if that modem is capable of a 9600 BPS connection. When using objects, the M parameter would be omitted.

See Also

SetLinkLocked

SetModemSpeed (APCOUR)

SetModemSpeed (APMCOM)

SetSpeedMatching

Syntax

```
procedure SetModemTimeouts(M : ModemRecPtr; Normal, Dialing : Integer);  
procedure AbstractModem.SetModemTimeouts(Normal, Dialing : Integer);
```

Purpose

Set timeout values for normal commands and dialing.

Description

This routine allows you to change the timeout values used when waiting for the modem to respond to dialing and answer commands (Dialing) and all other modem commands (Normal). The timeout values are expressed in clock ticks. The default values are 182 (DefTimeout = 10 seconds) for normal commands, and 1092 (DefDialTimeout = 60 seconds) for dialing and answer commands.

Example

```
SetModemTimeouts(M, 91, 546);
```

Wait 30 seconds ($30 \times 18.2 = 546$) for a response to a dialing command, 5 seconds ($5 \times 18.2 = 91$) for a response to a normal command. When using objects, the M parameter would be omitted.

See Also

GetModemResponse

Syntax

```
procedure SetModemVolume(M : ModemRecPtr; Opt : Word);  
procedure AbstractModem.SetModemVolume(Opt : Word);
```

Purpose

Set the speaker volume.

Description

This routine allows you to reset the volume control on the modem's speaker. Opt is one of the following: mLow (0), mMedium (1), or mHigh (2).

Although this command ('ATL') is not available for all modems, it is generally implemented on most internal modems and on external modems that lack a manual volume control.

To turn the speaker off completely, use SetModemSpeaker(mSpeakerOff).

Example

```
SetModemVolume(M, mLow);
```

Set the modem's volume control to its lowest setting. When using objects, the M parameter would be omitted.

See Also

SetModemSpeaker

Syntax

```
procedure AbstractModem.SetSpeedMatching; virtual;
```

Purpose

Answering DCE matches speed with originating DCE.

Description

This routine reverses the effects of SetModemSpeed. It instructs the modem to match whatever speed the remote modem is using. Since this is the default behavior of most modems, you need to call this routine only to cancel a previous call to SetModemSpeed.

This feature is available only on modems that support "managed modem links" such as MNP or V.42. If this feature is not supported by the modem object in use (HayesModem, for example), an ecNullCommand error is generated.

Example

```
SetModemSpeed(M, 9600);  
DialModem(M, '555-1111');  
{...transfer data and disconnect...}  
SetSpeedMatching(M);  
DialModem(M, '555-2222');
```

This example shows a program making two calls. The first call must be made at 9600 baud (to guarantee that large amounts of data can be transferred in a limited time). The second call will accept whatever speed is negotiated between the local and remote modems (because this call doesn't have large amounts of data to transfer).

See Also

SetModemSpeed
SetSpeedMatching (APCOUR)

SetSpeedMatching (APMCOM)

Store

Syntax

```
($IFDEF UseStreams)
procedure AbstractModem.Store(var S : IdStream);
```

Purpose

Store a modem object to a stream.

Description

Store writes an image of the modem object to the stream S.

S is a properly initialized stream object (a stream opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The stream registration routines for the modem object types are AbstractModemStream, HayesModemStream, CourierModemStream, MicrocomModemStream, NullModemStream, and AllModemStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses
  OpRoot, ApPort, OoCom, OoModem;
var
  UP : UartPort;
  HM : HayesModem;
  S : BufIdStream;
  Status : Word;
begin
  if not UP.InitFast(Com1, 2400) or
    not HM.Init(@UP) then begin
    WriteLn('Error initializing');
    Halt;
  end;

  if not S.Init('MODEM.STM', SCreate, 1024) then begin
    WriteLn('Error initializing stream');
    Halt;
  end;

  S.RegisterHier(AllModemStream);
  S.RegisterHier(UartPortStream);
  S.Put(UP);
  S.Put(HM);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error: ', Status);
    Halt;
  end;

  WriteLn('Store successful');
  S.Done;
  UP.Done;
  HM.Done;
end.
```

Instantiates the port and modem objects and then stores them in a stream using the stream's Put method, which calls the objects' Store methods indirectly.

See AbstractModem.Load for an example program that reloads these objects.

D. U.S. Robotics Courier Modem: APCOUR

This unit is required only if you are using the non-OOP calling format and are using a U.S. Robotics Courier 9600 BPS modem. Its sole purpose is to provide the command and response tables for that modem and the few extended procedures that can't be handled through the tables.

To take advantage of the features of APCOUR, include it in your uses list and call its InitModemCourier routine rather than InitModem in APMODEM. After that, you can still use all of APMODEM's standard features as well as APCOUR's extended features.

Declarations

Functions

The most important new routine this unit introduces is InitModemCourier, which initializes a ModemRecPtr to use the command and response tables for Courier 9600 modems.

Several other new procedures are shown below. These procedures work just like the functions of the same name in OOMODEM. Refer to the appropriate entry in §6.C for detailed information on how to use these methods.

```
procedure InitModemCourier(var M : ModemRecPtr; P : PortRecPtr);
  {-Allocate and initialize a Courier modem}
procedure SetDataCompression(M : ModemRecPtr; State : Boolean);
  {-Turn data compression on/off}
procedure SetErrorControl(M : ModemRecPtr; ErrorOpt : ErrorStates);
  {-Turn error control on/off}
procedure SetFlowControl(M : ModemRecPtr; FlowOpts : Word);
  {-Set flow control options}
procedure SetLinkLocked(M : ModemRecPtr; Locked : Boolean);
  {-Lock the link rate or follow connection?}
procedure SetModemSpeed(M : ModemRecPtr; Baud : LongInt);
  {-Set the DTE/DCE data rate}
procedure SetSpeedMatching(M : ModemRecPtr);
  {-Answering DCE matches speed with originating DCE}
```

E. Microcom Modem: APMCOM

This unit is required only if you are using the non-OOP calling format and are using a Microcom QX/3296c 9600 BPS modem. Its sole purpose is to provide the command and response tables for that modem and the few extended procedures that can't be handled through the tables.

To take advantage of the features of APMCOM, include it in your uses list and call its InitModemMcom routine rather than InitModem in APMODEM. After that, you can still use all of APMODEM's standard features as well as APMCOM's extended features.

Declarations

Functions

The most important new routine this unit introduces is InitModemMcom, which initializes a ModemRecPtr to use the command and response tables for Microcom modems.

Several other new procedures are shown below. These procedures work just like the functions of the same name in OOMODEM. Refer to the appropriate entry in §6.C for detailed information on how to use these methods.

```
procedure InitModemMcom(var M : ModemRecPtr; P : PortRecPtr);
  {-Allocate and initialize a Microcom modem}

procedure SetDataCompression(M : ModemRecPtr; State : Boolean);
  {-Turn data compression on/off}

procedure SetErrorControl(M : ModemRecPtr; ErrorOpt : ErrorStates);
  {-Turn error control on/off}

procedure SetFlowControl(M : ModemRecPtr; FlowOpts : Word);
  {-Set flow control options}

procedure SetLinkLocked(M : ModemRecPtr; Locked : Boolean);
  {-Lock the link rate or follow connection?}

procedure SetModemSpeed(M : ModemRecPtr; Baud : LongInt);
  {-Set the DTE/DCE data rate}

procedure SetSpeedMatching(M : ModemRecPtr);
  {-Answering DCE matches speed with originating DCE}
```

7. File Transfer Protocols

A. Overview

Many communications applications need to transfer files or other large amounts of data from one machine to another. Certainly, this could be done just by having the sender issue PutBlocks and the receiver issue GetBlocks. However, this would leave a tremendous amount of detail work to do. You'd have to add logic to transfer the file name and size information, check for and recover from transmission errors, handle the file I/O, and so on.

That's where file transfer protocols come in. The term "protocol" means that both sides of the communication link will behave in a clear, well-defined manner following agreed upon rules. The rules vary among the different protocols and some protocols offer more control and features than others. At a minimum, each protocol handles the file I/O and serial port I/O and checks for errors. The protocol might also include error correcting logic (by requesting retransmissions), file management rules, and recovery from partial file transfers.

Async Professional provides several industry standard file transfer protocols. The protocols are separated into different units—typically one unit includes one protocol and that protocol's extensions.

The APABSPCL/OOABSPCL units provide the abstract data and procedure declarations for all protocols. They also implement procedures that are common to all derived protocols (e.g., file I/O and status reporting). In fact, most of the routines you will use with protocols are defined in these units. You need to use one of these units in any program that uses protocols.

The APXMODEM/OOXMODEM units build on APABSPCL/OOABSPCL to provide the standard Xmodem protocol as well as some popular extensions: XmodemCRC, Xmodem1K and Xmodem1KG.

APYMODEM/OOYMODEM take the Xmodem protocol a bit further by adding file name and size information in a header block and by providing multi-file (batch) transfers.

APZMODEM/OOZMODEM provide the superb Zmodem protocol. Zmodem offers extremely fast file transfers with complete error control and file management facilities. Zmodem will generally be your first choice for file transfers.

APKERMIT/OOKERMIT offer the Kermit protocol. Kermit is a popular alternative, sometimes the *only* alternative, when transferring files between PCs and minicomputer or mainframe environments.

APBPLUS/OOBPLUS provide the CompuServe B+ protocol. This is a proprietary file transfer protocol designed and used exclusively by CompuServe.

APASCII/OOASCII are included in this chapter although ASCII isn't really a protocol. It offers a convenient means of transferring simple text files when file management and error control aren't important.

B. OOP vs. Non-OOP Protocols

The Async Professional protocol add-on layer continues the split started back at the interface layer. The OOP and non-OOP routines are completely separate and have separate source code files: all OOP files have a prefix of OO- (OOXMODEM) and all non-OOP files have a prefix of AP- (APXMODEM).

Just like the interface layer, the routines in the protocol add-on layer are nearly identical between the OOP and non-OOP formats. So again, they are documented only once, showing the appropriate declarations and pointing out the few differences. In the interface layer, the only consistent difference between the two formats was that the non-OOP format required a PortRecPtr. Similarly, the major consistent difference between the two formats of protocol routines is that the non-OOP format requires a ProtocolRecPtr.

The OOP and non-OOP protocols do differ just a bit more, though, in the area of procedure names. The OOP routines, by virtue of polymorphism, can use the same procedure names to initialize, process, and destroy all protocol types. The non-OOP routines can't do this.

A table illustrates this difference:

Purpose	OOP name	non-OOP name
initialize	Init	InitXxx (e.g. InitZmodem)
transmit	ProtocolTransmit	ProtocolTransmitXx (e.g. ProtocolTransmitZM)
receive	ProtocolReceive	ProtocolReceiveXx (e.g. ProtocolReceiveZM)
dispose	Done	DoneXxx (e.g. DoneZmodem)
misc.	SetFileMask	SetFileMask (no difference)

The OOP case is very simple. You always use Init, ProtocolTransmit, ProtocolReceive, and Done. The non-OOP case is a bit more complex, but once understood, it too is very simple. Each of these routines was given a name that indicates the protocol type (Xmodem, Ymodem, etc.). The convention is to make the protocol name part of the procedure name. For example, to initialize an Xmodem record, you call InitXmodem. The same is true when disposing of the protocol—to dispose of an Xmodem protocol, you call DoneXmodem.

Nearly the same is true for the calls to actually start the protocol: ProtocolReceive and ProtocolTransmit. However, only two letters of the protocol name were appended (since the procedure names were rather lengthy already). For example, to start up an Xmodem transmit, you call ProtocolTransmitXM.

Let's look at some code fragments to get an idea of the differences between OOP and non-OOP program:

```
OOP:
uses
  ApMisc, ApPort, ApUart, OoCom, OoAbsPcl, OoZmodem;
var
  ComPort : UartPortPtr;
  Protocol : ZmodemProtocolPtr;
...
{Instantiate a port}
New(ComPort, InitFast(Com3, 19200));
{...handle errors...}

{Instantiate a protocol}
New(Protocol, Init(ComPort));
{...handle errors...}
```



```

{Transmit a file}
Protocol^.SetFileMask('MAIL.LZH');
Protocol^.ProtocolTransmit;

non-OOP:
uses
  ApMisc, ApPort, ApUart, ApCom, ApAbsPcl, ApZmodem;
var
  ComPort : PortRecPtr;
  Protocol : ProtocolRecPtr;
...
{Instantiate a port}
InitPortFast(ComPort, Com3, 19200);
{...handle errors...}

{Instantiate a protocol}
InitZmodem(Protocol, ComPort);
{...handle errors...}

{Transmit a file}
SetFileMask(Protocol, 'MAIL.LZH');
ProtocolTransmitZM(Protocol);

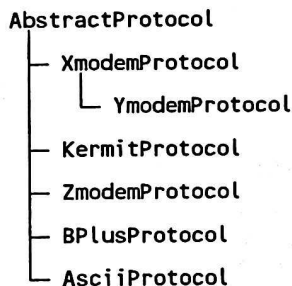
```

The OOP example has a UartPortPtr (ComPort) and a ZmodemProtocolPtr (Protocol). ComPort is instantiated first and is passed to Protocol's constructor. The example then transmits a file named MAIL.LZH.

The non-OOP example shows a PortRecPtr (ComPort) and a ProtocolRecPtr (Protocol). ComPort is initialized first by the call to InitPortFast. That ComPort pointer is passed to the InitZmodem routine, which returns an initialized Protocol pointer. The Protocol pointer is then passed to all protocol routines, as shown in the calls to SetFileMask and ProtocolTransmitZM.

The differences between the OOP and non-OOP format aren't inherently important (because you'll likely stick to only one format). However, understanding the differences will make the reference section and example programs much easier to follow.

If you are using protocol objects, here is the hierarchy you'll work with:



AbstractProtocol does all of the "boilerplate" work of a protocol. This includes the file I/O (reading data to transmit, and writing data that's received). It also has the routines for retrieving status information, managing the list of files to transmit, handling the various hooks that Async Professional provides, and controlling the various protocol options.

The derived protocols use these standard routines, but define and implement completely new routines for protocol-specific characteristics like the format of a block and error control.

C. Abstract Protocol Functions: APABSPCL/OOABSPCL

This unit serves several major functions. First, it provides the abstract data and procedure structure for all subsequent protocol types. The protocol object (or record) includes fields with information about the file being transferred (file name, size, etc.), information about the protocol (block size, block check method, etc.), and status information.

It also implements all procedures and functions that aren't dependent on the protocol type--setting the file transmit mask, setting protocol options, getting status information, and so on.

The AbstractProtocol object also provides the abstract procedures meant to be overridden and implemented by derived protocol objects.

The following short programs are two complete examples of how to transmit a file using the Zmodem protocol. First the OOP version:

```
program ExXfrO; {EXXFRO.PAS}
uses
  Crt, ApMisc, ApPort, ApUart, OoCom, OoAbsPcl, OoZmodem;
var
  ComPort : UartPortPtr;
  Protocol : ZmodemProtocolPtr;

procedure Abort(Msg : String; Code : Word);
begin
  WriteLn(Msg, ': ', Code);
  Halt;
end;

{$F+}
function KbdAbort : Boolean;
var
  C : Char;
begin
  KbdAbort := False;
  if KeyPressed then begin
    C := ReadKey;
    if C = #0 then C := ReadKey
    else if C = #$1B then KbdAbort := True;
  end;
end;

procedure ProtocolStatus(AP : AbstractProtocolPtr; Starting, Ending : Boolean);
begin
  if Starting then
    WriteLn(ProtocolTypeString[AP^.GetProtocol], ' started');

  WriteLn('Bytes transferred: ', AP^.GetBytesTransferred,
    ' Bytes remaining: ', AP^.GetBytesRemaining);

  if Ending then
    WriteLn(ProtocolTypeString[AP^.GetProtocol], ' ended');
end;
{$F-}

begin
  {Make port and protocol objects}
  New(ComPort, InitCustom(Com2, 9600, NoParity, 8, 1, 2048, 2048,
    DefPortOptions));
  if ComPort = nil then
```

```

    Abort('Failed to open port', AsyncStatus);
New(Protocol, Init(ComPort));
if Protocol = nil then
    Abort('Failed to initialize protocol', AsyncStatus);

{Set abort and status procedures}
ComPort^.SetAbortFunc(KbdAbort);
Protocol^.SetShowStatusProc(ProtocolStatus);

{Transmit one file}
Protocol^.SetFileMask('EXXFRO.PAS');
Protocol^.ProtocolTransmit;
if AsyncStatus = ecOk then
    WriteLn('Transfer completed')
else
    WriteLn('Transfer failed: ', AsyncStatus);

Dispose(Protocol, Done);
Dispose(ComPort, Done);
end.

```

This is a complete program for transmitting the file EXXFRO.PAS using the Zmodem protocol. It is a bit more than a minimal program would require since it takes advantage of two optional hooks: the abort hook and the status hook. Actually, *any* protocol program you write should include at least these two hooks so that the user can see what's going on and can cancel the protocol by pressing the <Esc> key. These two hooks are discussed later in this section.

The program first instantiates a port object (ComPort) and a protocol object (Protocol). It specifies the file to transmit by calling the method SetFileMask with the exact name of the file to transmit: EXXFRO.PAS. All of the work of actually transmitting the file is handled by ProtocolTransmit. It opens the file, handshakes with the remote computer, transfers the file following all the rules of the protocol, then cleanly ends the protocol with the remote computer and returns control to the example program. All that's left to do is check AsyncStatus to see what happened—whether the file was transferred successfully or there was an error.

Here's the non-OOP version of the same program:

```

program ExXfr; {EXXFR.PAS}
uses
    Crt, ApMisc, ApPort, ApUart, ApCom, ApAbsPcl, ApZmodem;
var
    ComPort : PortRecPtr;
    Protocol : ProtocolRecPtr;

procedure Abort(Msg : String; Code : Word);
begin
    WriteLn(Msg, ': ', Code);
    Halt;
end;

{$F+}
function KbdAbort : Boolean;
var
    C : Char;
begin
    KbdAbort := False;
    if KeyPressed then begin
        C := ReadKey;
        if C = #0 then C := ReadKey
        else if C = #$1B then KbdAbort := True;
    end;
end;

```

```

    end;
end;

procedure ProtocolStatus(P : ProtocolRecPtr; Starting, Ending : Boolean);
begin
    if Starting then
        WriteLn(ProtocolTypeString[GetProtocol(P)], ' started');

        WriteLn('Bytes transferred: ', GetBytesTransferred(P),
            ' Bytes remaining: ', GetBytesRemaining(P));

    if Ending then
        WriteLn(ProtocolTypeString[GetProtocol(P)], ' ended');
    end;
{$F-}

begin
    {Make port and protocol records}
    InitPort(ComPort, Com2, 9600, NoParity, 8, 1, 2048, 2048, DefPortOptions);
    if AsyncStatus <> ecOk then
        Abort('Failed to open port', AsyncStatus);
    InitZmodem(Protocol, ComPort);
    if AsyncStatus <> ecOk then
        Abort('Failed to initialize protocol', AsyncStatus);

    {Set abort and status procedures}
    SetAbortFunc(ComPort, KbdAbort);
    SetShowStatusProc(Protocol, ProtocolStatus);

    {Transmit one file}
    SetFileMask(Protocol, 'EXXFR.PAS');
    ProtocolTransmitZM(Protocol);
    if AsyncStatus = ecOk then
        WriteLn('Transfer completed')
    else
        WriteLn('Transfer failed: ', AsyncStatus);

    DoneZmodem(Protocol);
    DonePort(ComPort);
end.

```

This program is identical in function to the first. Notice, however, that it uses the non-OOP naming conventions when creating the protocol, transmitting the file, and when cleaning up.

Output Buffer Size

When transmitting files under protocol control, the output buffer size of the port *must* be at least 2078 bytes (2K plus 30 bytes). This is required because the protocols build the entire output buffer, including escaping, before sending the buffer to the port with PutBlockDirect. If you use 1024 byte blocks and every character is escaped, the output buffer needs to be 2048 bytes. The extra 30 bytes is a safety margin allowed so that the block check characters and the subsequent header, if present, can be added without a check for output buffer space. Note that there is no reason to use a buffer larger than 2078 bytes. In fact, a very large buffer could cause problems because it takes so long to drain.

When using the apZmodem8K option, the minimum output buffer size is 16414. Once again, there is no reason to use a larger buffer.

Aborting a Protocol

There will certainly be times when your program, or the person using it, needs to cancel a protocol transfer in progress (e.g., when something goes wrong at the remote computer or the user simply decides not to continue the transfer). All Async Professional protocols provide for this. They do not provide their own hook, however. Instead, they use the abort hook of the com port the protocol is using. If your program already has such a function in place for other com port activities, you don't need to do anything special to make it work with protocols. If your program doesn't already have an abort function, you will almost certainly want to add one to deal with protocols.

The typical abort function checks the keyboard for characters that your application uses to request an abort (e.g., <Esc> or <CtrlX>). Since most protocol transfers take place over modem links, you might also want to add a check for the modem DCD signal (data carrier detect). This way, you can abort a protocol quickly should the modem connection be broken.

See "User Abort Function" in §4.B for more information and examples.

Error Handling

All protocol transfers are subject to errors, including transmission errors (such as a parity error) as well as more mundane problems like file-not-found and out-of-disk-space.

Generally, your code to invoke the transfer will look something like this:

```
ZM".ProtocolTransmit;  
case AsyncStatus of  
  ecOK : {say everything was OK}  
  else  {report error}  
end;
```

Calls to ProtocolTransmit (or ProtocolReceive) return only after they have transmitted all files or failed due to a fatal error. Not all errors encountered during a protocol are fatal. In fact, the main purpose of the protocol is to detect and recover from as many errors as possible.

As with the abort hook, protocols do not use their own error handler. Instead, they pass all errors to the error procedure of the com port they are using. Protocols pass *all* errors to the error handler, including errors that they can recover from. This means that your error handler should generally ignore errors during protocol transfers. For example, you probably wouldn't want your error window to pop up saying you got a bad block or a parity error right in the middle of the protocol. These are conditions that are expected to occur during a protocol and are not really errors. In fact, the only reason the protocols call your error handler in such cases is that the information might prove useful while debugging your program.

See "User Error Procedure" in §4.B for more information and examples. See Appendix B for a list of all possible error codes and their meanings.

Status Procedure

ProtocolTransmit and ProtocolReceive handle all of the details of the protocol transfer from start to finish. This can take anywhere from a few seconds to several hours depending on how much data you transfer and how fast it is transferred. Since your program isn't in control during this time, it could be a bit difficult to figure out just what is going on. Is the protocol working? How many bytes have been transferred? These questions are difficult to answer with "black box" routines like ProtocolReceive and ProtocolTransmit.

Fortunately, Async Professional provides a great deal of visibility into the progress of a protocol transfer through the status procedure. During a protocol transfer, Async Professional makes

frequent calls to a status procedure that you provide. This gives you complete control over what information you want to display and how to display it.

The following shows the general form of a simple status procedure (see the example EXXFRO.PAS for a complete program that uses this routine):

```
{F+}
procedure ProtocolStatus(AP : AbstractProtocolPtr; Starting, Ending : Boolean);
begin
    if Starting then
        WriteLn(ProtocolTypeString[AP^.GetProtocol], ' started');
        WriteLn('Bytes transferred: ', AP^.GetBytesTransferred,
            ' Bytes remaining: ', AP^.GetBytesRemaining);
    if Ending then
        WriteLn(ProtocolTypeString[AP^.GetProtocol], ' ended');
end;
{F-}
...
Protocol^.SetShowStatusProc(ProtocolStatus);
```

This example shows the OOP calling sequence. The first parameter is a pointer to the protocol object that called the status routine. Use this pointer, as shown, to call the protocol's various status methods.

The non-OOP format for this routine is quite similar, simply replacing the AbstractProtocolPtr with a ProtocolRecPtr. Here's the declaration of a non-OOP status procedure:

```
procedure ProtocolStatus(P : ProtocolRecPtr; Starting, Ending : Boolean);
```

Other than this, the two routines behave exactly the same way.

This particular status procedure is exceedingly simple. It displays a startup message, one line with the current number of bytes transferred and number of bytes remaining and then an ending message when the protocol is over.

The Starting and Ending parameters are also useful when you need to build a popup window to display your status information. Starting is True only on the very first call to the status routine. It's your signal to build the window, save the underlying screen, and handle any other startup issues. Ending is True only on the very last call to the status routine. It's your signal to remove the window, restore the underlying screen, and handle any other cleanup issues.

The status routine is called once for each transmitted or received block. It is also called whenever the protocol has something new to report to the status routine (e.g., an error occurred or the receiver just received the file name). And if none of these conditions occur, the status function is called at least every 5 seconds.

The procedure SetShowStatusProc is used to install the status procedure. Each protocol can have a unique status procedure, or all protocols can share one status procedure. A valid status procedure must be declared FAR using {F+} or the far keyword, it must not be nested inside any other procedure, and it must take exactly the parameters shown in the examples above.

See FX.PAS and FXO.PAS for more realistic examples of windowed status procedures.

Logging Procedure

Very often, file transfers are automated. For example, your application might send all of the day's transaction files to a remote computer every night. In this case you would want to keep a record of files that were successfully transmitted and those that weren't.

This is where the Async Professional logging hook comes in. Its purpose is to give you an opportunity to log information about each received or transmitted file (whether the transfer succeeded or not). Your logging routine is called at the start and end of each file transfer. It is passed a code identifying the current logging action. Here's an example (see EXXFRO1.PAS and EXXFRI1.PAS for complete programs using this routine):

```
{ $F+ }
procedure LogFileActivity(AP : AbstractProtocolPtr;
                          LogFileStatus : LogFileType);
var
  FLog : Text;
begin
  Assign(FLog, 'EXAMPLE.HIS');
  Append(FLog);
  if IOResult = 2 then
    Rewrite(FLog);
  if IOResult <> 0 then
    Exit;

  case LogFileStatus of
    lfReceiveStart : WriteLn(FLog, AP^.GetFilename, ' receive start');
    lfReceiveOk    : WriteLn(FLog, AP^.GetFilename, ' receive ok');
    lfReceiveFail  : WriteLn(FLog, AP^.GetFilename, ' receive failed');
    lfReceiveSkip  : WriteLn(FLog, AP^.GetFilename, ' receive skipped');
    lfTransmitStart: WriteLn(FLog, AP^.GetFilename, ' transmit start');
    lfTransmitOk   : WriteLn(FLog, AP^.GetFilename, ' transmit ok');
    lfTransmitFail : WriteLn(FLog, AP^.GetFilename, ' transmit failed');
    lfTransmitSkip : WriteLn(FLog, AP^.GetFilename, ' transmit skipped');
  end;

  Close(FLog);
  if IOResult <> 0 then ;
end;
{ $F- }
...
Protocol^.SetLogFileProc(ProtocolLogging);
```

This example shows the OOP calling format. The non-OOP format is nearly identical, replacing the AbstractProtocolPtr with a ProtocolRecPtr. Here's the declaration of a non-OOP logging routine:

```
procedure LogFileActivity(P : ProtocolRecPtr; LogFileStatus : LogFileType);
```

Other than this change, the OOP and non-OOP logging routines work the same way.

This example shows every possible logging status value. See the entry for SetLogFileProc for information on the exact meaning of each one. For each call to the logging routine, the example appends an appropriate message to a history file called EXAMPLE.HIS. This particular message shows only the file name but, since you have access to the protocol pointer, more information is available (protocol name, file size, and so on).

Your logging routine isn't limited to logging information to a file. This is also your opportunity to take care of any file-related startup and cleanup activities. One example of this might be to delete

partially received files. You would probably want to do this for all protocols *except* Zmodem (since Zmodem can resume failed transfers at the failure point without having to retransmit the entire file).

SetFileLogProc is used to install the logging procedure. Each protocol can have a unique logging procedure, or all protocols can share one logging procedure. A valid logging procedure must be declared FAR using {\$F+} or the far keyword, it must not be nested inside any other procedure, and it must take exactly the parameters shown in the examples above.

See FX.PAS and FXO.PAS for additional examples of logging procedures.

NextFile Function

Several of the protocols provided by Async Professional can transmit and receive batches of files. Your program supplies the names of the files to transmit by implementing a NextFile function that the protocol calls.

In most cases, however, you won't need to worry about writing your own NextFile function because Async Professional has two built-in ones. NextFileMask is the default NextFile function used by all batch protocols. It works in conjunction with the SetFileMask routine to transmit all files that fit a particular mask. You can also use this method to transmit single files. Since this is the default NextFile function, your program only needs to call SetFileMask to take advantage of it:

```
New(Protocol, Init(ComPort));
Protocol^.SetFileMask('*.*.ZIP');
Protocol^.ProtocolTransmit;
```

This example instantiates Protocol, sets a file mask to include all files with the extension of 'ZIP', and then calls ProtocolTransmit to transmit the files. The same method could be used to transmit only one file:

```
Protocol^.SetFileMask('STUFF.TXT');
Protocol^.ProtocolTransmit;
```

In some cases a simple file mask might not be sufficient. For example, suppose you need to transmit two files: MAIL.ZIP and STUFF.TXT. There is no file mask (short of '*.*') that would include both of those files. Hence, Async Professional offers another alternative: NextFileList. This built-in function, in conjunction with several other built-in routines, helps you build a list of files to be transmitted. Here's how such a program might look:

```
var
  FLP : FileListPtr;
...
New(Protocol, Init(ComPort));
Protocol^.MakeFileList(FLP, 500);
Protocol^.AddFileToList(FLP, 'MAIL.ZIP');
Protocol^.AddFileToList(FLP, 'STUFF.TXT');
Protocol^.SetFileList(FLP);
Protocol^.SetNextFileFunc(NextFileList);
Protocol^.ProtocolTransmit;
Protocol^.DisposeFileList(FLP, 500);
```

FLP is a pointer to a FileList. A FileList is really just an array of characters. You must *always* call MakeFileList and tell it how many characters to use for this array. This is a packed array with only a separator character between each file name (if you're sizing it close to your actual required size, make sure you account for these separators). In this case, the array is 500 characters. At 13 characters per file (8 for the name, 1 for the '.', 3 for the extension, and 1 for the separator) 38 file names can be stored. Fewer file names can be stored if any of the names include a path.

Each file is added to the list with a call to AddFileToList. The list is attached to the protocol with a call to SetFileList. Finally, the call to SetNextFileFunc(NextFileList) tells the protocol to use the NextFileList.

Now, when ProtocolTransmit needs the next file name to transmit, NextFileList returns the next file in the list. When all the files in the list have been transmitted, ProtocolTransmit ends the protocol and returns control to your program.

These two NextFile functions should cover most of your needs. When they don't, you can always supply your own NextFile function and handle the situation yourself. For example, suppose you want to transmit files that fit several incompatible masks: '*.LZH', '*.ZIP', and '*.ARC'. With only the built-in functions at your disposal, you'd need to do this in three separate calls to ProtocolTransmit or expand the masks into a file list yourself. Neither task is difficult, but you might find it more convenient to write a NextFile function to do this for you.

For this particular problem, you'll find the FileMaskList object in OOARCHIV to be quite handy. See the programs EXXFRO3.PAS and EXXFR3.PAS for complete programs that show how to use the FileMaskList object in this situation. The custom NextFile function is shown below:

```
{F+}
function NextFileMaskList(AP : AbstractProtocolPtr;
                        var FName : PathStr) : Boolean;
    {-Custom function to compare all files in dir against list of masks}
const
    AnyFileButDir = AnyFile and not Directory;
var
    Finished : Boolean;
begin
    AsyncStatus := 0;
    FName := '';
    NextFileMaskList := False;
    Finished := False;

    {Loop through all files until we find one that meets a mask}
    repeat
        if AP^.FindingFirst then begin
            FindFirst('*. *', AnyFileButDir, AP^.CurRec);
            AP^.FindingFirst := False;
        end else
            FindNext(AP^.CurRec);

        {Check for errors}
        if DosError <> 0 then begin
            {Failed to find file, return error status}
            case DosError of
                3 : AP^.APort^.GotError(epFatal+ecDirNotFound);
                18 : Finished := True;
            end;
        end else if FML^.Match(AP^.CurRec.Name) then begin
            FName := AP^.CurRec.Name;
            NextFileMaskList := True;
            Finished := True;
        end;
    until Finished;
end;
{F-}
...
Protocol^.SetNextFileFunc(NextFileMaskList);
```

This example shows the OOP calling format. The non-OOP format is nearly identical, replacing the `AbstractProtocolPtr` with a `ProtocolRecPtr`. Here's the declaration of a non-OOP `NextFile` routine:

```
function NextFileMaskList(P : ProtocolRecPtr; var FName : PathStr) : Boolean;
```

Other than this change, the OOP and non-OOP `NextFile` routines work the same way.

This example relies on the internal protocol fields `FindingFirst` and `CurRec`. These particular fields are really for use by the built-in `NextFileMask` routine, but are handy here in the `FindFirst/FindNext` loop.

This example also relies on a global variable named `FML` (a `FileMaskListPtr`). Generally, a better approach is to include such a field in your own derived protocol; or use the `UserData` field of the object to hold a pointer to your data.

To find the next file, `NextFileMaskList` loops through all files in the current directory and passes each file name to the `FML^.Match` function. This function returns `True` if the passed file name matches any of the masks in its list. When a match is found, `FName` is set to the matching name and `NextFileMaskList` returns `True`. `ProtocolTransmit` then transmits that file.

`SetNextFileFunc` is used to install the `NextFile` function. Each protocol can have a unique `NextFile` function, or all protocols can share one `NextFile` function. A valid `NextFile` function must be declared `FAR` using `{$F+}` or using the `far` keyword, it must not be nested inside any other procedure, and it must take exactly the parameters shown in the examples above.

AcceptFile Function

This is the final user hook that Async Professional provides for protocols. Its purpose is to tell the protocol whether or not to accept a particular file from the remote, based on rules or conditions provided in your `AcceptFile` function. The `AcceptFile` function comes into play only when *receiving* files.

Of all the protocols that Async Professional provides, only Zmodem has any built-in file management rules. For example, you can instruct Zmodem to accept only files that don't already exist or that are newer than existing files (more on this in §7.F, "Zmodem Protocol"). This is fine for Zmodem, but the `AcceptFile` hook provides the capability to add similar rules to other protocols, or to extend Zmodem's own rules.

Once Async Professional knows the name of the file to receive, but before it starts receiving data, it calls your `AcceptFile` function. Your function should return `True` if it's OK to accept this file or `False` if this file should be rejected.

If your `AcceptFile` function returns `False`, Async Professional will not accept the file. Unfortunately, only the Zmodem protocol has provisions for skipping files. When your `AcceptFile` function rejects a file under Zmodem, Zmodem simply tells the remote computer to skip that file. The remote computer moves on to the next file or ends the protocol if there aren't any more files. Other protocols can't do this. For example, there's no way a Ymodem receiver can tell the remote transmitter to skip a file and move on to the next. So whenever your `AcceptFile` function rejects a file for any protocol but Zmodem, Async Professional has no choice but to cleanly abort the entire protocol. Although this seems a bit drastic, in practice it works quite well.

The `AcceptFile` function also provides an opportunity for you to rename a file if its current name isn't acceptable. For example, if the name of the incoming file conflicts with an existing file name, you might want to accept the file but change its name before writing it to disk.

Note that all Async Professional protocols already have options for handling incoming file name collisions. See WriteFailOptions in the Declarations in this section for a description. The file management rules and renaming abilities of the AcceptFile function are necessary only if the options provided by WriteFailOptions are inadequate.

Here's an example AcceptFile function that rejects all files with the .ARC extension (see EXXFR2.PAS and EXXFRO2.PAS for complete sample programs):

```

{$F+}
function ProtocolAccept(AP : AbstractProtocolPtr) : Boolean;
var
  S : String[13];
begin
  ProtocolAccept := True;
  S := AP^.GetFilename;
  DotPos := Pos('.', S);
  if DotPos <> 0 then begin
    S := Copy(DotPos+1, Length(S));
    if S = 'ARC' then
      ProtocolAccept := False;
  end;
end;
...
Protocol^.SetAcceptFileFunc(ProtocolAccept);

```

This example shows the OOP calling format. The non-OOP format is nearly identical, replacing the AbstractProtocolPtr with a ProtocolRecPtr. Here's the declaration of a non-OOP AcceptFile routine:

```

function ProtocolAccept(P : ProtocolRecPtr) : Boolean;

```

Other than this, the OOP and non-OOP AcceptFile routines work the same way.

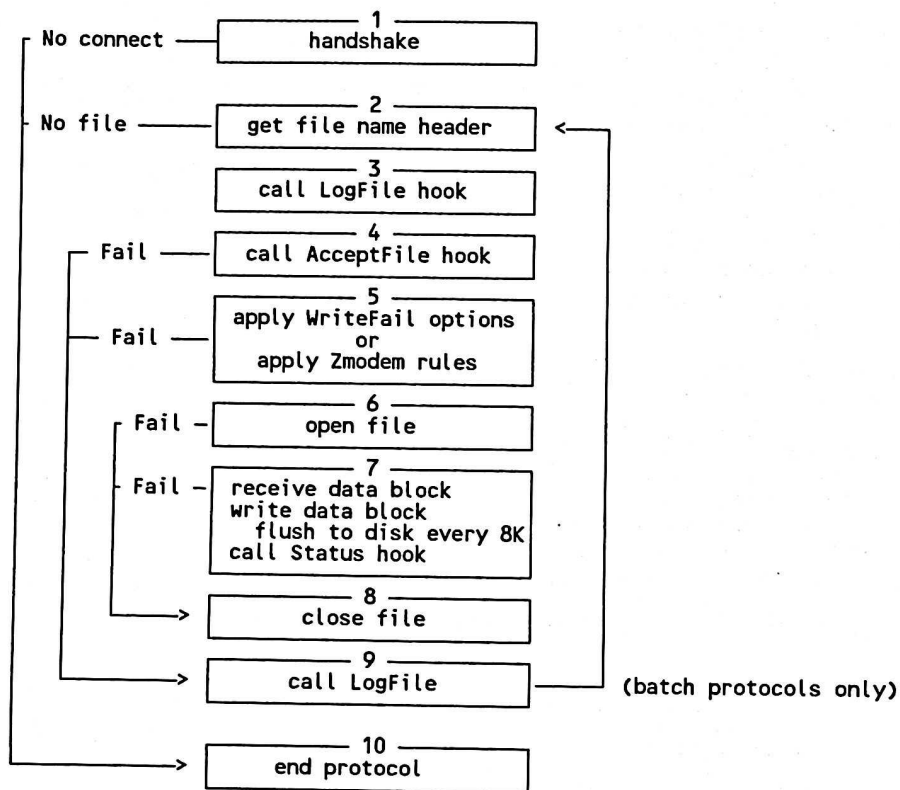
This is a rather simple example that returns True (accepts the file) in every case except where the file extension is ARC. The file name is converted to upper case before the AcceptFile function is called.

Async Professional offers one built-in AcceptFile function, AcceptOneFile, that is useful when you want to support batch protocols, but you *don't* want them to accept more than one file at a time. A common example might be a bulletin board system that supports Zmodem. Conceivably, someone might connect to your BBS and attempt to upload a multi-megabyte batch of files. Zmodem would continue to accept files until it ran out of disk space. To prevent this, you could restrict all uploads to one file (with the AcceptOneFile function) or you could write your own AcceptFile function to impose whatever restrictions are reasonable for your application.

Internal Logic

Until now, ProtocolTransmit and ProtocolReceive have been treated as black boxes—you set everything up ahead of time and call these routines to perform the protocol "magic." Now it is time to look at a broad overview of just how these two routines work. Given a general understanding of the order in which things are done, you should be better able to understand not only how to use the protocols, but also how to best take advantage of the hooks they provide.

When receiving files, ProtocolReceive employs the following logic. This diagram generally applies to all protocols.



On entry, ProtocolReceive attempts to "handshake" with the remote machine. If the handshake is unsuccessful after the specified number of retries, the protocol ends.

If the handshake is successful, the transmitter is asked for the name of the next file to transmit. It responds by sending a block containing the name.

Next the LogFile routine is called to give the application an opportunity to record the file name or take any other special action needed at the start of the transfer. The LogFile routine is called before the AcceptFile function so that it can log the original name before AcceptFile has a chance to change it.

The AcceptFile routine is called next. This is the first point at which the protocol might fail. If the AcceptFile function returns False (meaning you don't want this file) the file is skipped and control is transferred to the ending LogFile call.

Next the built-in WriteFail options are applied or Zmodem's built-in file management rules are applied. Again, if the protocol fails at this point, control is transferred to the ending LogFile call.

The file is created using the name from the file name header, perhaps as modified by the AcceptFile function. Step 6 also allocates work buffers and sets up several internal variables to keep track of buffering (Async Professional always uses an 8K internal file buffer). If the open should fail, control is transferred to step 8, the "close file" step which disposes of buffers and closes the file.

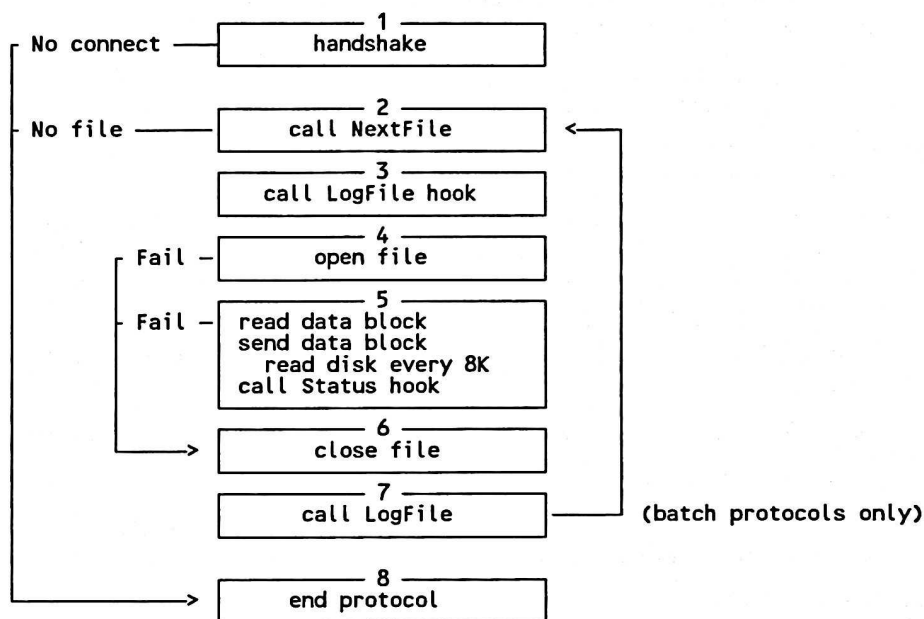
Step 7, the actual transfer of data, comes next. The internals of this loop vary tremendously among the protocols, so it is condensed in this diagram. Note that the status routine is called at least once

for each block received. Should this step fail for any reason (user abort, broken connection, disk full, etc.), control is transferred to step 8 again.

Then the LogFile hook is called again. This time the appropriate ending information (whether the file was received OK, skipped, or failed) is passed in.

In a batch protocol, the next step is to loop back up to the top and try to get another file header. If so, the whole process is repeated. Otherwise, the protocol is over and all that remains is to end gracefully, coordinating with the transmitter, and to clean up.

When transmitting files, ProtocolTransmit employs the following logic. This diagram also generally applies to all protocols.



On entry, ProtocolTransmit attempts to "handshake" with the remote machine. If the handshake is unsuccessful after the specified number of retries, the protocol ends.

If the handshake is successful, the NextFile function is called to get the name of the next file to transmit. If there are no more files to transmit, the protocol is over.

The LogFile routine is called to give the application a chance to log the file name or take care of any other application-specific startup activities.

Then the file is opened. This step also allocates work buffers and initializes buffer variables. Should any of these steps fail, control is transferred to "close file" to clean up.

If the file was successfully opened, transmission is started by reading data from the disk and sending data blocks to the remote computer. As when receiving data, Async Professional uses 8K buffers for buffering the file data. The Status procedure is called at least once for every block.

Should any fatal errors occur in this step, control is transferred to step 6 to close the file and dispose of working buffers.

Step 7, the ending call to LogFile, is executed whether the transfer was successful or not (the appropriate code is passed to your logging routine). If this is a batch protocol, control is transferred

to step 2 to get another file to transmit. If there are no more files, or this is not a batch protocol, the protocol ends and clean up is performed.

There is one very important consideration when transmitting files under protocol control--the output buffer size of the port *must* be at least 2078 bytes (which is 2K plus 30 bytes). This is required because the protocols build the entire output buffer, including escaping, before sending the buffer to the port with PutBlockDirect. If you are using 1024 byte blocks and every character is escaped, then the output buffer must hold 2048 bytes. The extra 30 bytes is a safety margin allowed so that the block check characters and the subsequent header, if there is one, don't need to check for output buffer space.

The protocols use this method primarily for performance reasons--it's always faster to build a block and call PutBlockDirect than it is to call PutChar for each character in the block.

File Read/Write Routines

APABSPCL and OOABSPCL include internal file I/O routines that are called by the derived protocols to write received data or to read data to transmit. In nearly all cases you'll find it acceptable to let these routines handle the file I/O for you.

In a few rare circumstances you might need to modify the standard behavior (such as compressing and decompressing data "on the fly" or transmitting data that doesn't originate from a DOS file). Async Professional provides for these possibilities by isolating all file I/O within six routines, three each for input and output, and giving you the means to supply your own routines.

These routines are briefly described here, though you should study their implementations before attempting your own.

In OOABSPCL, file input operations are handled by the following three virtual methods:

```
procedure apPrepareReading; virtual;
function apReadProtocolBlock(var Block : DataBlockType;
                             var BlockSize : Word) : Boolean; virtual;
procedure apFinishReading; virtual;
```

The following code fragment shows how the protocols use these methods:

```
apPrepareReading;
if {OK} then begin
  repeat
    if apReadProtocolBlock(Block, Size) then
      {transmit block to remote};
    until {error or end-of-file};
  apFinishReading;
end;
```

AbstractProtocol's implementation of these three methods assumes that the input is a standard DOS file. apPrepareReading opens the file specified by the NextFile function. apReadProtocolBlock is called repeatedly, asking for the next BlockSize bytes from the file, until end-of-file or a fatal error occurs.

Should you need to alter any of these steps, you would derive a new protocol object and override one or more of these methods. Suppose, for example, that your program is generating data that it needs to transmit to another machine via Zmodem. You could generate the data, write it to a file, and then transmit that file.

A faster approach might be to derive a new protocol from ZmodemProtocol and override all three of the above methods. Each time your apReadProtocolBlock method is called, it would return the

next BlockSize bytes of generated data. Your apPrepareReading and apFinishReading methods would do whatever start up and clean up work is associated with generating the data.

The same concept applies when writing data received from a protocol. In this situation, the three virtual methods are:

```

procedure apPrepareWriting; virtual;
function apWriteProtocolBlock(var Block : DataBlockType;
                             BlockSize : Word) : Boolean; virtual;
procedure apFinishWriting; virtual;

```

The derived protocols use these methods as follows:

```

apPrepareWriting;
if {OK} then begin
  repeat
    {receive a block of data from remote};
    Finished := apWriteProtocolBlock(Block, Size);
  until {Finished or fatal error};
  apFinishWriting;
end;

```

AbstractProtocol's implementation of these methods assumes that the data is to be written to a standard DOS file. apPrepareWriting opens the file using the file name supplied by the transmitter (or as modified by the AcceptFile function). apWriteProtocolBlock writes each block of data as it is received and apFinishWriting closes the file. Should you need to alter any of AbstractProtocol's standard processing, you would derive you own protocol and override one or more of these methods.

If you are using the non-OOP protocols, you obviously can't use virtual methods to change the file I/O handling. Instead, APABSPCL uses the following three procedure types and six procedure pointer fields:

```

type
  PrepFinishProc = procedure(P : ProtocolRecPtr);
  ReadProtProc = function(P : ProtocolRecPtr;
                          var Block : DataBlockType;
                          var BlockSize : Word) : Boolean;

  WriteProtProc = function(P : ProtocolRecPtr;
                          var Block : DataBlockType;
                          BlockSize : Word) : Boolean;

  ProtocolData = record
    ...
    PrepareReading      : PrepFinishProc;    {Proc for preparing file reads}
    ReadProtocolBlock   : ReadProtProc;       {Proc for reading blocks}
    FinishReading       : PrepFinishProc;     {Proc for closing read file}
    PrepareWriting      : PrepFinishProc;     {Proc for preparing file writes}
    WriteProtocolBlock  : WriteProtProc;      {Proc for writing blocks}
    FinishWriting       : PrepFinishProc;     {Proc for closing write file}
    ...
  end;

```

The file I/O routines must be declared FAR using {\$F+} or using the far keyword, they must not be nested inside any other procedure, and they must take exactly the parameters shown in the appropriate procedure type definition.

The use of these routines is exactly the same as for the virtual methods described earlier. The only difference is in the way the procedure pointers are set. They are part of an internal record of general protocol data and are set with code like this:

```
var
  ZM : ProtocolRecPtr;
...
  InitProtocolZM(ZM);
  {handle errors}
  ZM^.PData^.PrepareReading := MyPrepareProc;
  ZM^.PData^.ReadProtocolBlock := MyBlockProc;
  ZM^.PData^.FinishReading := MyFinishProc;
...
```

Background File Transfer

The implementation of the file transfer protocols in Async Professional provides a unique feature: the ability to transfer files in the "background." In this context, background means that the protocol functions can perform a small increment of work and return control to your program. With such a capability, your application can transfer files while doing other things and you can even build memory resident programs that transfer files in the background while your PC is running another application. Naturally, all of the protocols continue to operate exactly as expected.

During a file transfer protocol, your program actually spends much of its time waiting—waiting for incoming data, waiting for an acknowledgement or, in streaming protocol transmits, waiting for the output buffer to drain. Most protocol implementations don't make very good use of this surplus of processing time. With Async Professional, your program can regain control whenever the protocols are in one of their wait states.

It is easiest to see how the protocols, and your program, achieve this by contrasting the protocol calling process described previously with the background calling process. As shown previously, your program starts a protocol with code like this:

```
var
  ZM : ZmodemProtocolPtr;
begin
  New(ZM, Init(ComPort));
  ZM^.SetFileMask('*.');
  ZM^.ProtocolTransmit;
  Dispose(ZM, Done);
end.
```

This program instantiates a Zmodem object, sets the file mask to all files, and calls ProtocolTransmit to transmit the files.

Using the background protocol functions, this program would be changed as follows:

```
var
  ZM : ZmodemProtocolPtr;
begin
  New(ZM, Init(ComPort));
  ZM^.SetFileMask('*.');
  ZM^.PrepareTransmitPart;
  repeat
    until ZM^.ProtocolTransmitPart = psFinished;
  Dispose(ZM, Done);
end.
```

{<---- new}
{<---- new}
{<---- new}

The call to `ProtocolTransmit` was replaced with three new statements: a call to `PrepareTransmitPart` and a repeat loop that calls `ProtocolTransmitPart` until it returns a result of `psFinished`. The call to `PrepareTransmitPart` sets various fields in the protocol object to the correct initial values; each call to `ProtocolTransmitPart` performs the next increment of required work.

Called in a tight repeat loop like this, `ProtocolTransmitPart` often has nothing to do. It is waiting to receive data or, in this specific example, it is usually waiting for the output buffer to drain so that it can transmit the next block of data. If there is some work that it can perform, it performs just that slice of work and exits again.

Obviously, the place for you to achieve multitasking is within the repeat loop. Your program can do many different things here as long as you keep in mind the guideline that the activity should be fairly short—executing lengthy routines slows down the protocol. Put another way, if the condition the protocol was waiting for occurs before you call `ProtocolTransmitPart` again, you increase the overall time required to transmit the file. To assure the protocol proceeds at its best possible rate, you should always call `ProtocolTransmitPart` as frequently as possible.

Naturally, if the foreground process is more important to you than the ultimate speed of the file transfer, then you can discount this requirement and call `ProtocolTransmitPart` at whatever reasonable frequency you desire. (We say "reasonable frequency" because delaying too long in calling `ProtocolTransmitPart` could cause the remote machine to timeout or even abort the file transfer.)

Notice that `ProtocolTransmitPart` is a function that returns an enumerated type. There are three possible values in this enumeration: `psReady`, `psWaiting`, and `psFinished`.

`psReady` means that the protocol is immediately ready to perform its next action. Your application could use this value to skip its other processing and call `ProtocolTransmitPart` again immediately. `psWaiting` means that the protocol is currently waiting for some condition. Unfortunately, it's not possible to reliably predict how long the protocol will be in this "wait state." However, this is the best opportunity for your application to perform some small amount of other work. `psFinished` means that the protocol is finished, either correctly or with errors (examine `AsyncStatus` to determine which). This is the only value you must check for, as the above example does, so that you'll know when to stop calling `ProtocolTransmitPart`.

The User Background Hook

Depending on the type of work your application needs to perform while the protocol is in progress, you might find it convenient to continue to use the regular protocol functions (`ProtocolTransmit` and `ProtocolReceive`) while specifying a user background hook. Using this method, your background routine is constantly called whenever the protocol routine itself returns. The same restrictions described above still apply, however. Your user background procedure should perform only small increments of work and exit as soon as possible. The background procedure is called only when the protocol is in a `psWaiting` state, so your background procedure can always assume it's safe to do some work.

Below is a typical user background function:

```
{F+}
procedure MyBackgroundProc(AP : AbstractProtocolPtr);
var
  C : Char;
begin
  if KeyPressed then begin
    C := ReadKey;
    case C of
```

```

        cEsc : PendingAbort := True;
        {otherwise handle the keystroke as the application requires}
    end;
end;
end;
{$F-}
...
ZM^.SetBackgroundProc(MyBackgroundProc);

```

This particular example doesn't do much except show what a background procedure should look like. Your background procedure can do anything it wants (except serial activity using the same port) so long as it does it quickly.

By default, the procedure `NoUserBack` is defined as the user background procedure. This is an empty procedure that does nothing.

The `UserBackProc` is defined somewhat differently for the OOP and non-OOP interfaces. Here are the definitions:

OOP:

```

{Procedure called whenever protocol is in psWaiting state}
UserBackProc = procedure(AP : AbstractProtocolPtr);

```

non-OOP:

```

{Procedure called whenever protocol is in psWaiting state}
UserBackProc = procedure(P : ProtocolRecPtr);

```

The only difference is that the OOP format is passed an `AbstractProtocolPtr` whereas the non-OOP format receives a `ProtocolRecPtr`.

`SetBackgroundProc` is used to install your `UserBackProc`. Each protocol can have its own `UserBackProc` or all protocol objects can use the same one.

A valid `UserBackProc` procedure must be declared FAR using `{$F+}` or the far keyword, must not be nested in any other procedure, and must take exactly the parameters shown in the example above.

One special consideration that may arise is what to do if both your background procedure and your user abort procedure need to process keyboard input. Generally, you can't have more than one user hook processing keyboard input at one time. If both of these procedures call `KeyPressed` and `ReadKey`, each may confuse the other since they are called at different frequencies and may read and discard keyboard input intended for the other.

The preferred approach, illustrated in part above, is to move the keyboard abort checking logic to the user background hook and use a global boolean variable to communicate the abort request to the user abort function. So, for the user background function shown above, the companion user abort function looks something like this:

```

function MyAbort : Boolean;
    {-Aborts work-in-progress}
var
    C : Char;
begin
    if ComPort^.ProtocolInProgress then
        MyAbort := PendingAbort
    else begin
        if KeyPressed then begin

```

```

    MyAbort := (ReadKey = cEsc);
    if C = #0 then
        C := ReadKey;
    end;
end;
end;
end;

```

Notice that this abort function still checks the keyboard for an <Esc> if a protocol isn't in progress, since the user abort function is also used to abort from Wait I/O routines and WaitForChar routines.

Multitasking a Protocol

"Multitasking a protocol" means that your program is doing other work, visible or not, while the protocol is in progress. Examples include processing an editor or other user-interface object, processing other protocols on other ports, processing a communications terminal on another port, and so on.

Depending on the type of multitasking you need to do, you may choose one of three approaches:

- 1) Using ProtocolTransmit/ProtocolReceive and a user background hook to perform your multitasking work.
- 2) Calling PrepareXxxPart and ProtocolXxxPart in your own repeat loop, performing your multitasking work within the loop.
- 3) Keeping another object or portion of your program in control and having it call ProtocolXxxPart whenever it has time.

The method you choose will depend on the other work you want to perform. If you simply need to update a clock on the screen or do some other trivial multitasking, you can use method 1) or 2). As the complexity of the task increases, you may find it more convenient to use method 2), since it gives you more control over the scope of variables.

For multitasking with a user-interface process like an editor or data entry screen, you'll probably need to switch your thinking a bit and call ProtocolXxxPart from within the background hook of *that* process. For example, all Object Professional user-interface objects provide a background hook (while checking for or reading keystrokes). To multitask an Object Professional Editor object with a protocol, you would call ProtocolXxxPart from within the Editor's background hook whenever there were no keystrokes waiting to be processed.

Using Protocols in a TSR

Another capability of the protocol architecture is the ability to transfer files in the background from a memory resident program. This doesn't mean a TSR that pops up and transfers a file while popped up, but a TSR that transfers the file in the background while your machine is running another program in the foreground. For example, you could be in the middle of your word processor, popup the TSR, start the protocol, popdown, and return to your editor. The file transfer would continue in the background while you worked within your editor.

Async Professional doesn't provide any of the TSR or ISR routines necessary for creating a memory resident program. This discussion, and the example programs, assume that you are using the TSR routines from Object Professional, TSRs Made Easy, or Turbo Professional. However, any TSR routines with similar capabilities should work as well.

A TSR protocol program will probably require several keyboard popup entry points: one for starting and stopping the protocol, one for displaying status information and, perhaps, one for a general purpose terminal. The only one of these entry points that requires special mention here is the status entry point. Since status information will be requested on demand from the keyboard, it is not convenient to use the usual status hook provided by the protocols (since you can't accurately predict how often it will be called). Hence, you'll probably find it easier just to call the protocol's

information routines directly from your status popup entry point and display the information there. You would employ the following logic in your status function:

```
prepare for screen writing
set a timer
do
    if timer expired then begin
        reset timer
        get protocol information
        display protocol information
    end
while not keypressed
cleanup screen writing
```

The objective is to keep the status display window on the screen, updating it at some reasonable interval, until the user presses any key.

Your TSRs will also need some method of "waking up" every so often to call either ProtocolTransmitPart or ProtocolReceivePart. This is typically done by taking over the clock interrupt (\$1C) and counting how many ticks go by. When the desired number of ticks (at 18.2 ticks per second) elapse, your program should call ProtocolTransmitPart or ProtocolReceivePart. Note, however, that you can't call either of these routines directly from the clock interrupt—they take too long and they may require DOS services. So, you'll need to schedule another routine, using the services of your TSR library, that can use DOS services and *that* routine will be the one that calls ProtocolTransmitPart or ProtocolReceivePart. If you are using TSR routines from TurboPower Software products, SetPopTicker is the scheduling routine that your clock interrupt must call.

See the demonstration programs BACK/BACKO in §7.J for an example of how to use protocols in a TSR.

Shared Declarations

Constants

apIncludeDirectory	= \$0001;	{True to include directory in file names}
apHonorDirectory	= \$0002;	{True to honor directory in file names}
apRTSLowForWrite	= \$0004;	{True to lower RTS during disk writes}
apKermitNoStripName	= \$0008;	{True to not strip Kermit filenames}
apKermitDisplay	= \$1000;	{True to honor KDisplay packets}
apKermitLongPackets	= \$2000;	{True to support long packets}
apKermitSWC	= \$4000;	{True to support SWC}
apZmodem8K	= \$8000;	{True to allow support for 8K blocks}

Option codes for protocols. The apIncludeDirectory option, meaningful only when transmitting files, tells the protocol to send the remote the complete pathname for the file being transmitted. The apHonorDirectory option, meaningful only when receiving files, tells the protocol to try to honor any path specified in the name of a file being received. (Note, however, that no attempt will be made to create a directory that doesn't already exist.)

Typically, apIncludeDirectory and apHonorDirectory are used only when you are writing both of the programs involved in the transmission (the sender and the receiver) and the sender needs to be able to force the receiver to store files in a specific directory. You do not use these options when communicating with a BBS or information service, for example.

apRTSLowForWrite is off by default. If enabled via apOptionsOn(apRTSLowForWrite), all Async Professional protocols force RTS off while writing received data to disk, temporarily preventing the modem from sending additional data. RTS is turned back on as soon as the disk write is finished. This option might be required when running Async Professional programs under some multi-tasking operating systems or in network environments that leave interrupts disabled while writing to network devices.

apKermitNoStripName is off by default. In this state, the Kermit protocol strips directory information from file names when transmitting the file name packet. If you want the directory information included in the filename packet, set this option on. Note, however, that many Kermit implementations do not expect to find directory information in the file name and don't behave predictably when they do find it.

If apKermitDisplay is set, APKERMIT/OOKERMIT write the contents of Kermit KDisplay packets to the screen using a WriteLn statement. Kermit KDisplay packets are used by some hosts to return an error or status message after a file transfer. The WriteLn statement displays the message at the current cursor location using the current TextMode attribute. This is probably inappropriate for most windowed user interfaces, so apKermitDisplay is off by default.

apKermitLongPackets is an "information only" option and is off initially. It is turned on if you make a successful call to SetMaxLongPacketLen. Although it is primarily intended for internal use, you may want to reference this option in your protocol status display.

apKermitSWC is used to indicate whether Kermit Sliding Windows Control is enabled or not. It is turned on if you make a successful call to SetMaxWindows. Although it is primarily intended for internal use, you may want to reference this option in your protocol status display.

apZmodem8K is off by default. Turn this option on to use Zmodem with 8K data blocks. See APZMODEM/OOZMODEM in §7.F for details on 8K data block support and usage.

BadProtocolOptions : Word = apKermitLongPackets+apKermitSWC+apZmodem8K;

Protocol options intended strictly for internal use by Async Professional. BadProtocolOptions is set to the three options that must be off when a protocol is instantiated.

```
bcNone      = 0; {No block checking}
bcChecksum1 = 1; {Basic checksum}
bcChecksum2 = 2; {Two byte checksum}
bcCrc16     = 3; {16 bit CRC}
bcCrc32     = 4; {32 bit CRC}
bcCrck      = 5; {Kermit style CRC}
```

Values returned by GetCheckType to describe the type of block checking being used by the current protocol.

```
bcsNone      = 'No check ';
bcsChecksum1 = 'Checksum ';
bcsChecksum2 = 'Checksum2';
bcsCrc16     = 'Crc16    ';
bcsCrc32     = 'Crc32    ';
bcsCrck      = 'CrcKermit';
```

Descriptive strings corresponding to the values bcNone through bcCrck. Intended for use in status routines.

BlockFillChar : Char = ^Z;

Character used to fill the last block transmitted by a protocol that doesn't keep track of file size.

DefHandshakeRetry = 10;

Default number of times to retry when handshaking fails. See SetHandshakeWait.

DefHandshakeWait = 182;

Default time to wait (in clock ticks) for a response during handshaking (182 clock ticks = 10 seconds). See SetHandshakeWait.

DefProtocolOptions : Word = 0;

Default protocol options. Currently there are no such options.

DefStatusInterval = 91;

The status function is called every time a block is sent or received. If no blocks are sent or received within DefStatusInterval ticks, the status function is called anyway (91 ticks = 5 seconds).

DefTransTimeout = 1092;

Ticks to wait for output buffer space (1092 ticks = 60 seconds).

FileBufferSize = 8192;

Size of the working buffer used for receiving or transmitting files.

GmtHourOffset : Byte = 0;

The difference, in hours, between the current time zone and Greenwich Meridian Time (GMT). The specifications for Ymodem and Zmodem state that the file creation time stamp should be referenced to GMT. To meet this specification, both the receiver and the transmitter must adjust the time stamp by the difference between the current time zone and the GMT time zone. In practice, it seems that most Ymodem/Zmodem implementations ignore this offset.

ProtocolTypeString : array[XModem..UserProt3] of String[10] = (
 'Xmodem', 'XmodemCRC', 'Xmodem1K', 'Xmodem1KG', 'Ymodem', 'YmodemG',
 'Zmodem', 'Kermit', 'Ascii', 'BPlus', 'UserProt1', 'UserProt3', 'UserProt3');

An array of strings describing all supported protocols. Useful in status routines.

TelixonDelay : Byte = 9;

Testing shows that when using Async Professional Zmodem with Telix's Zmodem, a delay is needed between files (9 ticks seems to work well). If you are not using Telix, set TelixonDelay to 0 for slightly faster multi-file transfers.

Xmodem = 0;

XmodemCRC = 1;

Xmodem1K = 2;

Xmodem1KG = 3;

Ymodem = 4;

YmodemG = 5;

Zmodem = 6;

Kermit = 7;

Ascii = 8;

BPlus = 9;

UserProt1 = 10;

UserProt2 = 11;

UserProt3 = 12;

These are the currently supported protocol types. UserProt1 through UserProt3 can be used by your application for new protocols that you implement.

Types

DataBlockType = array[1..1024] of Char;

Data structure used for storing received and transmitted blocks.

FileBufferArray = array[0..FileBufferSize-1] of Byte;

Working buffer used for receiving or transmitting files.

FileListPtr = ^FileListType;

FileListType = array[0..65535-1] of Char;

Array used to hold a list of files to be transmitted.

LogFileType = (

 lfReceiveStart, lfReceiveOk, lfReceiveFail, lfReceiveSkip,

 lfTransmitStart, lfTransmitOk, lfTransmitFail, lfTransmitSkip);

Parameter passed to a LogFile procedure to describe an event to be logged. See SetLogFileProc.

WriteFailOptions = (WriteFail, WriteRename, WriteAnyway, WriteResume);

Describes an action to be taken if a file to be received already exists. See SetOverwriteOption.

```

ProtocolStateType = (
    psReady,           {Ok to call again immediately}
    psWaiting,         {Protocol is waiting, ok to do something else}
    psFinished);       {Protocol is finished}

```

A result of this type is returned by the functions ProtocolTransmitPart and ProtocolReceivePart to indicate the current state of the protocol.

APABSPCL Declarations

Types

```
AcceptFileFunc = function(P : ProtocolRecPtr) : Boolean;
```

A function called by a protocol when receiving a file to allow the application to accept or reject the file based on its own criteria.

```
LogFileProc = procedure(P : ProtocolRecPtr; LogFileStatus : LogFileType);
```

A procedure called by a protocol for the purpose of logging file transfer events for auditing purposes.

```
NextFileFunc = function(P : ProtocolRecPtr; var FName : PathStr) : Boolean;
```

A function called by a batch protocol to return the name of the next file to transmit.

```
ProtocolDataPtr = ^ProtocolData;
```

```
ProtocolData =
```

```
    record
```

```
    ...
```

```
    end;
```

Data needed by the routines in APABSPCL to work with all supported protocols (file name and size, status information, etc).

```
ProtocolRecPtr = ^ProtocolRec;
```

```
ProtocolRec =
```

```
    record
```

```
        PData : ProtocolDataPtr;
```

```
    end;
```

Generic protocol record containing only a pointer to the ProtocolData record described above. This is the pointer type you will use with all protocols. Internally, the protocols typecast this to the appropriate type (XmodemPtr, YmodemPtr, etc.). The typecast pointer types generally include additional fields specific to the protocol.

```
ShowStatusProc = procedure(P : ProtocolRecPtr; Starting, Ending : Boolean);
```

A routine of this type is used to display status information on the screen while uploading or downloading files using a protocol.

```
UserBackProc = procedure(P : ProtocolRecPtr);
```

A procedure of this type can be specified as the user background hook within a protocol. Such a procedure is called by the protocol as frequently as possible (generally, whenever the protocol is in a psWaiting state).

OOABSPCL Declarations

Types

```
AbstractProtocolPtr = ^AbstractProtocol;
```

```
AbstractProtocol =
```

```
    object(Root)
```

```
    ...
```

```
    end;
```

Abstract object that defines data fields and methods to be shared in common by higher-level protocol objects.

AcceptFileFunc = function(AP : AbstractProtocolPtr) : Boolean;

A function called by a protocol when receiving a file to allow the application to accept or reject the file based on its own criteria.

LogFileProc = procedure(AP : AbstractProtocolPtr; LogFileStatus : LogFileType);

A procedure called by a protocol for the purpose of logging file transfer events for auditing purposes.

**NextFileFunc = function(AP : AbstractProtocolPtr;
var FName : PathStr) : Boolean;**

A function called by a batch protocol to return the name of the next file to transmit.

**ShowStatusProc = procedure(AP : AbstractProtocolPtr;
Starting, Ending : Boolean);**

A routine of this type is used to display status information on the screen while uploading or downloading files using a protocol.

UserBackProc = procedure(AP : AbstractProtocolPtr);

A procedure of this type can be specified as the user background hook within a protocol. Such a procedure is called by the protocol as frequently as possible (generally, whenever the protocol is in a psWaiting state).

AddFileToList / apOptionsAreOn / apOptionsOff

Syntax

```
procedure AddFileToList(P : ProtocolRecPtr; FLP : FileListPtr; PName : PathStr);  
procedure AbstractProtocol.AddFileToList(FLP : FileListPtr; PName : PathStr);
```

Purpose

Add filename to a file list.

Description

This routine adds the specified filename (PName) to a file list (FLP) previously created by a call to MakeFileList. If there is not sufficient space in the list, an ecOutOfMemory error is reported in AsyncStatus.

Important: PName should name a single file, already known to exist on the sender's machine. It should not contain any DOS wildcard characters.

Example

See the example for SetFileList.

See Also

DisposeFileList
MakeFileList

SetFileList

Syntax

```
function apOptionsAreOn(P : ProtocolRecPtr; OptionFlags : Word) : Boolean;  
function AbstractProtocol.apOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose

Return True if all specified options are on.

Description

This function returns True if the specified protocol option(s) are currently selected.

Example

```
if apOptionsAreOn(P, apHonorDirectory) then  
  apOptionsOff(P, apHonorDirectory)  
else  
  apOptionsOn(P, apHonorDirectory);
```

Toggle the state of the apHonorDirectory option. When using objects, the P parameters would be omitted.

Syntax

```
procedure apOptionsOff(P : ProtocolRecPtr; OptionFlags : Word);  
procedure AbstractProtocol.apOptionsOff(OptionFlags : Word);
```

Purpose

Deactivate multiple options.

Description

This procedure deactivates the specified protocol option(s), excluding those designated as BadProtocolOptions.

Example

See the example for apOptionsAreOn.

apOptionsOn / DisposeFileList / Done

Syntax

```
procedure apOptionsOn(P : ProtocolRecPtr; OptionFlags : Word);  
procedure AbstractProtocol.apOptionsOn(OptionFlags : Word);
```

Purpose

Activate multiple options.

Description

This procedure activates the specified protocol option(s), excluding those designated as BadProtocolOptions.

Example

See the example for apOptionsAreOn.

See Also

apOptionsAreOn

apOptionsOff

Syntax

```
procedure DisposeFileList(P : ProtocolRecPtr; FLP : FileListPtr; Size : Word);  
procedure AbstractProtocol.DisposeFileList(FLP : FileListPtr; Size : Word);
```

Purpose

Dispose of a file list.

Description

This routine disposes of a file list (FLP) of the specified Size. The file list should have been previously created by a call to MakeFileList.

Example

See the example for SetFileList.

See Also

AddFileToList
MakeFileList

SetFileList

Syntax

```
destructor AbstractProtocol.Done; virtual;
```

Purpose

Dispose of a protocol object.

Description

This Done method disposes of the internal data structures allocated on the heap by the Init method. It is an internal method, and should be called only by the destructor for a protocol object whose immediate parent is AbstractProtocol.

EstimateTransferSecs / GetBlockErrors

Syntax

```
function EstimateTransferSecs(P : ProtocolRecPtr; Size : LongInt) : LongInt;  
function AbstractProtocol.EstimateTransferSecs(Size : LongInt)  
    : LongInt; virtual;
```

Purpose

Return the estimated time to transfer the specified number of bytes.

Description

This function is intended to be used in status routines to obtain an estimate of the time required (in seconds) to transfer Size bytes of data. Typically, a status routine calls this function with two different parameters, the total size of the file (from GetFileSize) and the number of bytes remaining to be transferred (from GetBytesRemaining).

Example

```
var  
    TotalTime, TimeLeft : LongInt;  
...  
    if Starting then begin  
        TotalTime := EstimateTransferSecs(P, GetFileSize);  
        ...  
    end;  
    ...  
    TimeLeft := EstimateTransferSecs(P, GetBytesRemaining);
```

Presumably part of a status routine. EstimateTransferSecs is called once when the transmission starts to calculate the amount of time required to transfer the entire file. It is thereafter called repeatedly to calculate the amount of time remaining. For good examples of how to use this routine, see the WindowStatus routine in FX.PAS and FXO.PAS.

See Also

GetBytesRemaining
GetFileSize

SetEfficiencyParms

Syntax

```
function GetBlockErrors(P : ProtocolRecPtr) : Word;  
function AbstractProtocol.GetBlockErrors : Word;
```

Purpose

Return the number of errors received during this block.

Description

This function is intended to be used in status routines to determine how many errors occurred in the transmission of the current block.

Example

```
WriteLn('BlockErrors: ', GetBlockErrors(P));
```

Presumably part of a status routine. Displays the current number of block errors. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS for additional examples.

See Also

GetTotalErrors

GetBlockNum / GetBlockSize

Syntax

```
function GetBlockNum(P : ProtocolRecPtr) : Word;  
function AbstractProtocol.GetBlockNum : Word;
```

Purpose

Return the current block number.

Description

This function is intended to be used in status routines to determine the number of the current block. For more information on the block numbers returned by each protocol, see the PROTD0C.LZH archive.

Example

```
WriteLn('Block number: ', GetBlockNum(P));
```

Presumably part of a status routine. Displays the current block number. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS for additional examples.

See Also

GetBlockSize

Syntax

```
function GetBlockSize(P : ProtocolRecPtr) : Word;  
function AbstractProtocol.GetBlockSize : Word;
```

Purpose

Return the current block size.

Description

This function is intended to be used in status routines to determine the size of the current block.

Example

```
WriteLn('Block size: ', GetBlockSize(P));
```

Presumably part of a status routine. Displays the current size of each block. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS for additional examples.

See Also

GetBlockNum

GetFileSize

GetBytesRemaining / GetBytesTransferred

Syntax

```
function GetBytesRemaining(P : ProtocolRecPtr) : LongInt;  
function AbstractProtocol.GetBytesRemaining : LongInt;
```

Purpose

Return the number of bytes not yet transferred.

Description

This function is intended to be used in status routines to determine the number of bytes remaining to be transferred. The amount of time required to transfer the remaining data can be estimated by EstimateTransferSecs.

Example

```
WriteLn('Bytes remaining: ', GetBytesRemaining(P));
```

Presumably part of a status routine. Displays the number of bytes remaining to be transferred. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS and the example for EstimateTransferSecs for additional examples.

See Also

EstimateTransferSecs
GetBytesTransferred

GetFileSize

Syntax

```
function GetBytesTransferred(P : ProtocolRecPtr) : LongInt;  
function AbstractProtocol.GetBytesTransferred : LongInt;
```

Purpose

Return the number of bytes already transferred.

Description

This function is intended to be used in status routines to determine the number of bytes of data already transmitted.

Example

```
WriteLn('Bytes transferred: ', GetBytesTransferred(P));
```

Presumably part of a status routine. Displays the number of bytes already transferred. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS for additional examples.

See Also

GetBytesRemaining

GetElapsedTics

GetCheckType / GetCurrentBlockNum

Syntax

```
function GetCheckType(P : ProtocolRecPtr) : Byte;  
function AbstractProtocol.GetCheckType : Byte;
```

Purpose

Return the code for the block check type.

Description

This function is intended to be used in status routines to determine the type of block checking being used (16-bit CRC, 32-bit CRC, etc.) by the protocol. The value returned is in the range bcNone through bcCrkK.

Example

```
var  
  S : string;  
...  
  case GetCheckType(P) of  
    bcNone      : S := bcsNone;  
    bcChecksum1 : S := bcsChecksum1;  
    bcChecksum2 : S := bcsChecksum2;  
    bcCrk16     : S := bcsCrk16;  
    bcCrk32     : S := bcsCrk32;  
    bcCrkK      : S := bcsCrkK;  
  end;
```

This code fragment shows how to convert the value returned by GetCheckType to a string. For good examples of how to use this routine, see the WindowStatus routine in FX.PAS and FXO.PAS.

See Also

GetProtocol

Syntax

```
function GetCurrentBlockNum(P : ProtocolRecPtr) : Word;  
function AbstractProtocol.GetCurrentBlockNum : Word;
```

Purpose

Return the number of the block currently being transferred.

Description

Use this function instead of GetBlockNum if you want the number of the block currently being transferred rather than the number of the last completely transferred block.

Note that all Async Professional example and demonstration programs use GetBlockNum.

See Also

GetBlockNum

Syntax

```
function GetElapsedTics(P : ProtocolRecPtr) : LongInt;  
function AbstractProtocol.GetElapsedTics : LongInt;
```

Purpose

Return the number of clock ticks since the first block was sent (or received).

Description

This function is intended to be used in status routines to determine the amount of time elapsed (in clock ticks) since the first block in the file was sent/received.

Example

```
WriteLn('Elapsed time: ', Tics2Secs(GetElapsedTics(P)));
```

Presumably part of a status routine. Displays, in seconds, the elapsed time for the current file. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS for additional examples.

See Also

GetBytesTransferred

Syntax

```
function GetFileName(P : ProtocolRecPtr) : PathStr;  
function AbstractProtocol.GetFileName : PathStr;
```

Purpose

Return the name of the current file.

Description

This function is intended to be used in status routines to obtain the name of the file currently being transferred. Unlike GetPathname, GetFileName never returns a drive or directory, only the name of the file.

When the file name is not available, GetFileName returns an empty string. During batch protocol receives (YModem, ZModem, Kermit), the name of the file is not available the first few times your protocol status routine is called. This is because it may take a few seconds before the transmitter sends the file name packet.

Example

```
WriteLn('File name: ', GetFilename(P));
```

Presumably part of a status routine. Displays the name of the file currently being transferred. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS for additional examples.

See Also

GetPathname

GetFileSize / GetInitialFilePos

Syntax

```
function GetFileSize(P : ProtocolRecPtr) : LongInt;  
function AbstractProtocol.GetFileSize : LongInt;
```

Purpose

Return the current file size (0 if no file active).

Description

This function is intended to be used in status routines to determine the size of the file currently being transmitted. If no file is being transmitted, GetFileSize returns 0.

Note that not all protocols know the size of an incoming file (Xmodem, Kermit, and ASCII do not). In these cases, GetFileSize returns 0.

Example

```
WriteLn('File size: ', GetFileSize(P));
```

Presumably part of a status routine. Displays the size of the file currently being transferred. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS and the example for EstimateTransferSecs for additional examples.

See Also

EstimateTransferSecs
GetBytesRemaining

GetBytesTransferred

Syntax

```
function GetInitialFilePos(P : ProtocolRecPtr) : LongInt;  
function AbstractProtocol.GetInitialFilePos : LongInt;
```

Purpose

Return the initial file position during resumed file transfers.

Description

When a resumed file transfer is in progress, the function GetBytesTransferred returns the *total* number of bytes transferred for the file: bytes transferred during the interrupted session plus bytes transferred during the current session. If you use this value to calculate your transfer rate, it will be too high because it includes bytes already transferred. The simple equation:

```
CPS := GetBytesTransferred(P) div Trunc(CurElapsedTics(P) / 18.2);
```

won't work anymore. Instead, you need to subtract out the bytes that were transferred in the previous session (i.e., the starting file size):

```
BytesThisSession := GetBytesTransferred(P) - GetInitialFilePos(P);
```

```
CPS := BytesThisSession div Trunc(CurElapsedTics(P) / 18.2);
```

This function can be used for all protocols, because zero is returned when the current transfer is not a resumed transfer or the current protocol doesn't support resumed file transfers.

You won't always want to subtract GetInitialFilePos from GetBytesTransferred. The status functions in the example programs (FX/FXO) only do so when calculating CPS. When displaying the "bytes transferred" field, the total bytes transferred for a file is displayed rather than just the bytes transferred this session.

Zmodem protocol has the ability to resume interrupted file transfers. B+ protocol has the ability to resume interrupted file receives. If you aren't using Zmodem or B+, you won't need to use this function.

See Also

GetBytesTransferred

Syntax

```
function GetPathname(P : ProtocolRecPtr) : PathStr;
function AbstractProtocol.GetPathname : PathStr;
```

Purpose

Return the complete pathname of the current file (if known).

Description

This function is intended to be used in status routines to obtain the name of the file currently being transferred. Unlike GetFilename, GetPathname sometimes returns a drive and/or directory, as well as the name of the file. It can also be used in AcceptFile routines to determine the name of the file about to be received.

Note that GetPathName will *not* return the pathname if the pathname was never specified to the protocol (either in a file name block or by the NextFile function). If you want to determine the complete pathname when all you know is the file name, use a function like the DOS unit's FExpand.

When the file name is not available, GetPathName returns an empty string. During batch protocol receives (YModem, ZModem, Kermit), the name of the file is not available the first few times your protocol status routine is called. This is because it may take a few seconds before the transmitter sends the file name packet.

Example

```
{ $F+ }
procedure ProtocolStatus(AP : AbstractProtocolPtr;
    Starting, Ending : Boolean);
    {-Custom show status procedure}
begin
    ...
    FastWrite(AP^.GetPathname,...);
    ...
end;
function MyNextFileMask(AP : AbstractProtocolPtr;
    var FName : PathStr) : Boolean;
    {-Built-in function that works with file mask fields}
begin
    {return complete pathname of next file to transmit}
end;
{ $F- }
...
ZM^.SetNextFileFunc(MyNextPathName);
ZM^.SetShowStatus(ProtocolStatus);
...
```

Although the bulk of the code has been omitted, this example shows where a protocol status routine might use GetPathname instead of GetFilename. Notice that the NextFile function returns the complete pathname. If the NextFile function omitted the path, then GetPathname would omit it also.

See Also

GetFilename
SetAcceptFileFunc

SetShowStatusProc

GetProtocol / GetTotalErrors

Syntax

```
function GetProtocol(P : ProtocolRecPtr) : Byte;  
function AbstractProtocol.GetProtocol : Byte;
```

Purpose

Return the current protocol type.

Description

This function is intended to be used in status routines to determine the protocol currently being used to transfer the file. The value returned is in the range Xmodem through Ascii. The value can be converted to a string using the typed constant array ProtocolTypeString.

Example

```
var  
  S : string;  
...  
  S := ProtocolTypeString[GetProtocol(P)];
```

Demonstrates how to get a string representation of the name of the protocol currently in use.

See Also

GetCheckType

SupportsBatch

Syntax

```
function GetTotalErrors(P : ProtocolRecPtr) : Word;  
function AbstractProtocol.GetTotalErrors : Word;
```

Purpose

Return the number of errors received during this transfer.

Description

This function is intended to be used in status routines to determine the total number of errors that have occurred since a file transfer began.

Example

```
WriteLn('Total errors: ', GetTotalErrors(P));
```

Presumably part of a status routine. Displays the number of protocol errors that have been encountered since the protocol started transferring the current file. When using objects, the P parameter would be omitted. See the WindowStatus routine in FX.PAS and FXO.PAS for additional examples.

See Also

GetBlockErrors

Syntax

```
constructor AbstractProtocol.InitCustom(AP : AbstractPortPtr; Options : Word);
```

Purpose

Allocate and initialize a protocol control block.

Description

This constructor initializes the data fields common to all objects derived from AbstractProtocol. It is an internal method, and should be called only by the constructor for a protocol object whose immediate parent is AbstractProtocol. AP is a pointer to the port object to be used. Options is a bit-mapped word indicating the desired protocol options (typically DefProtocolOptions).

Syntax

```
procedure MakeFileList(P : ProtocolRecPtr; var FLP : FileListPtr; Size : Word);  
procedure AbstractProtocol.MakeFileList(var FLP : FileListPtr; Size : Word);
```

Purpose

Allocate a new file list.

Description

This routine tries to allocate a block of memory of the specified Size that will store the names of files to be transferred using a batch protocol. If successful, FLP returns a pointer to the block of memory. If not, an ecOutOfMemory error is reported in AsyncStatus.

Example

See the example for SetFileList.

See Also

AddFileToList
DisposeFileList

SetFileList
SetNextFileFunc

PrepareReceivePart

Syntax

```
procedure PrepareReceivePartXM(P : ProtocolRecPtr);
procedure PrepareReceivePartYM(P : ProtocolRecPtr);
procedure PrepareReceivePartZM(P : ProtocolRecPtr);
procedure PrepareReceivePartKM(P : ProtocolRecPtr);
procedure PrepareReceivePartBP(P : ProtocolRecPtr);
procedure PrepareReceivePartAS(P : ProtocolRecPtr);
procedure AbstractProtocol.PrepareReceivePart; virtual;
```

Purpose

Prepare to call the background routine ProtocolReceivePart.

Description

This procedure must be called before the initial call to ProtocolReceivePart. It sets various internal fields to their proper initial values. You also need to call this procedure if you want to start a subsequent protocol transfer session (after the first one is finished).

A non-OOP version of this procedure does not exist in APABSPCL, even though six non-OOP declarations are shown above. PrepareReceivePartXM is found in APXMODEM; PrepareReceivePartYM is found in APYMODEM; etc. These functions are documented here because they work exactly the same as PrepareReceivePart.

Examples

```
var
  ZM : ProtocolRecPtr;
begin
  InitZmodem(ZM, ComPort);
  PrepareReceivePartZM(ZM);
  repeat
    until ProtocolReceivePartZM(ZM) = psFinished;
  DoneZmodem(ZM);
end.
```

PrepareReceivePartZM is called before entering the main repeat loop to initialize the protocol state table to its initial state.

```
var
  ZM : ZmodemProtocolPtr;
begin
  New(ZM, Init(ComPort));
  ZM^.PrepareReceivePart;
  repeat
    until ZM^.ProtocolReceivePart = psFinished;
  Dispose(ZM, Done);
end.
```

This is the OOP equivalent of the previous example.

```
var
  ZM : ZmodemProtocolPtr;
begin
  New(ZM, Init(ComPort));
  ZM^.PrepareReceivePart;
  ZM^.ProtocolReceive;
end.
```

Incorrect! (but harmless) You don't need to call PrepareReceivePart unless you will be calling ProtocolReceivePart directly. Since this example is calling ProtocolReceive to process the protocol, the call to PrepareReceivePart is unnecessary.

See Also

AbstractProtocol.ProtocolReceivePart

Syntax

```

procedure PrepareTransmitPartXM(P : ProtocolRecPtr);
procedure PrepareTransmitPartYM(P : ProtocolRecPtr);
procedure PrepareTransmitPartZM(P : ProtocolRecPtr);
procedure PrepareTransmitPartKM(P : ProtocolRecPtr);
procedure PrepareTransmitPartBP(P : ProtocolRecPtr);
procedure PrepareTransmitPartAS(P : ProtocolRecPtr);

procedure AbstractProtocol.PrepareTransmitPart; virtual;

```

Purpose

Prepare to use the background routine ProtocolTransmitPart.

Description

This procedure must be called before the initial call to ProtocolTransmitPart. It sets various internal fields to their proper initial values. You also need to call this procedure if you wish to start a subsequent protocol transfer session (after the first one is finished).

A non-OOP version of this procedure does not exist in APABSPCL, even though six non-OOP declarations are shown above. PrepareTransmitPartXM is found in APXMODEM; PrepareTransmitPartYM is found in APYMODEM; etc. These functions are documented here because they work exactly the same as PrepareTransmitPart.

Examples

```

var
  ZM : ProtocolRecPtr;
begin
  InitZmodem(ZM, ComPort);
  PrepareTransmitPartZM(ZM);
  SetFileMask(ZM, '*.');
  repeat
    until ProtocolTransmitPartZM(ZM) = psFinished;
  DoneZmodem(ZM);
end.

```

PrepareTransmitPartZM is called before entering the main repeat loop to initialize the protocol state table to its initial state.

```

var
  ZM : ZmodemProtocolPtr;
begin
  New(ZM, Init(ComPort));
  ZM^.PrepareTransmitPart;
  ZM^.SetFileMask '*.');
  repeat
    until ZM^.ProtocolTransmitPart = psFinished;
  Dispose(ZM, Done);
end.

```

This is the OOP equivalent of the previous example.

```

var
  ZM : ZmodemProtocolPtr;
begin
  New(ZM, Init(ComPort));
  ZM^.SetFileMask '*.');
  ZM^.PrepareTransmitPart;
  ZM^.ProtocolTransmit;
end.

```

Incorrect! (but harmless) You don't need to call PrepareTransmitPart unless you will be calling ProtocolTransmitPart directly. Since this example is calling ProtocolTransmit to process the protocol, the call to PrepareTransmitPart is unnecessary.

ProtocolReceive

Syntax

```
procedure ProtocolReceiveXM(P : ProtocolRecPtr);
procedure ProtocolReceiveYM(P : ProtocolRecPtr);
procedure ProtocolReceiveZM(P : ProtocolRecPtr);
procedure ProtocolReceiveKM(P : ProtocolRecPtr);
procedure ProtocolReceiveBP(P : ProtocolRecPtr);
procedure ProtocolReceiveAS(P : ProtocolRecPtr);
procedure AbstractProtocol.ProtocolReceive; virtual;
```

Purpose

Start protocol receive.

Description

This routine should be called when a program is ready to receive a file (or files, if the protocol in use supports batch file transfers) being transmitted to it via a protocol. Control is returned to the calling program only when all files are received successfully or a fatal error aborts the transmission. If the program needs to display status information concerning the transmission in progress, it should use the ShowStatus hook (see SetShowStatusProc).

A non-OOP version of this routine does not exist in APABSPCL, even though six non-OOP declarations are shown above. ProtocolReceiveXM is found in APXMODEM; ProtocolReceiveYM is found in APYMODEM; etc. These functions are documented here because they work exactly the same as ProtocolReceive.

Example

```
var
  ZM : ProtocolRecPtr;
...
  InitZmodem(ZM, ComPort);
  if AsyncStatus <> ecOK then
    {handle error};
  ProtocolReceiveZM(ZM);
```

Initializes a ZModem protocol and receives a batch of files.

```
var
  ZM : ZmodemProtocolPtr;
...
  New(ZM, Init(ComPort));
  if ZM = nil then
    {handle error};
  Protocol.ProtocolReceive;
```

Instantiates a Zmodem protocol object and receives a batch of files.

See the TransferFiles routine in FXO.PAS for a more comprehensive example.

See Also

SetShowStatusProc

Syntax

```
function ProtocolReceivePartXM(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolReceivePartYM(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolReceivePartZM(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolReceivePartKM(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolReceivePartBP(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolReceivePartAS(P : ProtocolRecPtr) : ProtocolStateType;

function AbstractProtocol.ProtocolReceivePart : ProtocolStateType ; virtual;
```

Purpose

Perform one increment of a background protocol receive.

Description

Call this function as often as possible and as many times as required. The protocol is finished when this function returns psFinished. At that time your program can examine AsyncStatus to determine whether the protocol finished successfully or with errors.

If the function result is psReady, then the protocol is ready to do additional work and should be called again as soon as possible. You don't have to call the protocol again immediately, however. Failing to do so will usually just increase the amount of time it takes to transfer the file. Don't delay too long, though, or the remote computer may decide that you aren't responding and cancel the protocol.

If the function result is psWaiting, then the protocol is waiting for some condition (usually for the next block of data) and it's relatively safe for your program to do other work before calling ProtocolReceivePart again. Unfortunately, it's impossible for the protocol to tell you exactly how much time you have (since the protocol can't really know when the next block of data will be ready). You can probably make some educated guesses based on block size and baud rate, but even those guesses will be thrown off by network delays or flow control.

A non-OOP version of this function does not exist in APABSPCL, even though six non-OOP declarations are shown above. ProtocolReceivePartXM is found in APXMODEM; ProtocolReceivePartYM is found in APYMODEM; etc. These functions are documented here because they work exactly the same as ProtocolReceivePart.

Example

```
var
  ZM : ProtocolRecPtr;
begin
  InitZmodem(ZM, ComPort);
  PrepareReceivePartZM(ZM);
  repeat
    until ProtocolReceivePartZM(ZM) = psFinished;
  DoneZmodem(ZM);
end.
```

ProtocolReceivePartZM is called repeatedly until it returns a function result of psFinished.

```
var
  ZM : ZmodemProtocolPtr;
begin
  New(ZM, Init(ComPort));
  ZM^.PrepareReceivePart;
  repeat
    until ZM^.ProtocolReceivePart = psFinished;
  Dispose(ZM, Done);
end.
```

This is the OOP equivalent of the previous example.

See Also

AbstractProtocol.PrepareReceivePart

AbstractProtocol.ProtocolTransmitPart

ProtocolTransmit

Syntax

```
procedure ProtocolTransmitXM(P : ProtocolRecPtr);
procedure ProtocolTransmitYM(P : ProtocolRecPtr);
procedure ProtocolTransmitZM(P : ProtocolRecPtr);
procedure ProtocolTransmitKM(P : ProtocolRecPtr);
procedure ProtocolTransmitBP(P : ProtocolRecPtr);
procedure ProtocolTransmitAS(P : ProtocolRecPtr);
procedure AbstractProtocol.ProtocolTransmit; virtual;
```

Purpose

Start protocol transmit.

Description

This routine should be called when a program is ready to transfer a file (or files, if the protocol in use supports batch file transfers) to the remote via a protocol. Control is returned to the calling program only when all files are sent successfully or a fatal error aborts the transmission. If the program needs to display status information concerning the transmission in progress, it should use the ShowStatus hook (see SetShowStatusProc).

A non-OOP version of this routine does not exist in APABSPCL, even though six non-OOP declarations are shown above. ProtocolTransmitXM is found in APXMODEM; ProtocolTransmitYM is found in APYMODEM; etc. These functions are documented here because they work exactly the same as ProtocolTransmit.

Example

```
var
  ZM : ProtocolRecPtr;
...
  InitZmodem(ZM, ComPort);
  if AsyncStatus <> ecOK then
    {handle error};
  SetFileMask(ZM, '*.*');
  ProtocolTransmitZM(ZM);
```

Initializes a ZModem protocol and transmits all files in the current directory.

```
var
  ZM : ZmodemProtocolPtr;
...
  New(ZM, Init(ComPort));
  if ZM = nil then
    {handle error};
  Protocol^.SetFileMask('*.*');
  Protocol^.ProtocolTransmit;
```

Instantiates a Zmodem protocol object and transmits all files in the current directory.

See the TransferFiles routine in FXO.PAS for a more comprehensive example.

See Also

SetShowStatusProc

Syntax

```
function ProtocolTransmitPartXM(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolTransmitPartYM(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolTransmitPartZM(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolTransmitPartKM(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolTransmitPartBP(P : ProtocolRecPtr) : ProtocolStateType;
function ProtocolTransmitPartAS(P : ProtocolRecPtr) : ProtocolStateType;
function AbstractProtocol.ProtocolTransmitPart : ProtocolStateType ; virtual;
```

Purpose

Perform one increment of a background protocol transmit.

Description

Call this function as often as possible and as many times as required. The protocol is finished when this function returns psFinished. At that time your program can examine AsyncStatus to determine whether the protocol finished successfully or with errors.

If the function result is psReady, then the protocol is ready to do additional work and should be called again as soon as possible. You don't have to call the protocol again immediately, however. Failing to do so will usually just increase the amount of time it takes to transfer the file. Don't delay too long, though, or the remote computer may decide that you aren't responding and cancel the protocol.

If the function result is psWaiting, then the protocol is waiting for some condition (usually for an acknowledgement or for free space in the output buffer) and it's relatively safe for your program to do other work before calling ProtocolTransmitPart again. Unfortunately, it's impossible for the protocol to tell you exactly how much time you have.

A non-OOP version of this function does not exist in APABSPCL, even though six non-OOP declarations are shown above. ProtocolTransmitPartXM is found in APXMODEM; ProtocolTransmitPartYM is found in APYMODEM; etc. These functions are documented here because they work exactly the same as ProtocolTransmitPart.

Examples

```
var
  ZM : ProtocolRecPtr;
begin
  InitZmodem(ZM, ComPort);
  PrepareTransmitPartZM(ZM);
  repeat
    until ProtocolTransmitPartZM(ZM) = psFinished;
  DoneZmodem(ZM);
end.
```

ProtocolTransmitPartZM is called repeatedly until it returns a function result of psFinished.

```
var
  ZM : ZmodemProtocolPtr;
begin
  New(ZM, Init(ComPort));
  ZM^.PrepareTransmitPart;
  repeat
    until ZM^.ProtocolTransmitPart = psFinished;
  Dispose(ZM, Done);
end.
```

This is the OOP equivalent of the previous example.

See Also

AbstractProtocol.PrepareTransmitPart

AbstractProtocol.ProtocolReceivePart

SetAcceptFileFunc

Syntax

```
procedure SetAcceptFileFunc(P : ProtocolRecPtr; AFP : AcceptFileFunc);  
procedure AbstractProtocol.SetAcceptFileFunc(AFP : AcceptFileFunc);
```

Purpose

Set a function to be called when a file is received.

Description

This routine lets you specify a user-written function (AFP) that will be called just before a file is received from the remote, giving you an opportunity to accept or reject the file. More specifically, this hook allows you to

- reject specific files for file management reasons (e.g., because the file already exists)
- impose restrictions on number/size of files sent at the other end (e.g., you might want to prevent the user from sending more than N files, or from sending more than N bytes of data)
- rename incoming files under your own specific rules (as opposed to the general ones controlled by calling SetOverwriteOption)

Do keep in mind when using this hook that, unless you are using Zmodem, rejecting a file will abort the protocol.

If using APABSPCL, an AcceptFile function should be of the following form:

```
{SF+}  
function AcceptFile(P : ProtocolRecPtr) : Boolean;  
begin  
    AcceptFile := True;  
    ...  
end;
```

P is a pointer to the data used to process the protocol. This parameter can be passed to GetPathname to obtain the name of the file about to be received. The function should return True to accept the file, or False to reject it.

If using OOABSPCL:

```
{SF+}  
function AcceptFile(AP : AbstractProtocolPtr) : Boolean;  
begin  
    AcceptFile := True;  
    ...  
end;
```

The name of the file to be received can be obtained by calling AP^.GetPathname.

Both APABSPCL and OOABSPCL provide two built-in AcceptFile functions, AcceptOneFile, which simply insures that no more than one file is received, regardless of whether or not a batch file transfer was initiated by the remote, and NoAcceptFile, which always returns True. NoAcceptFile is used by default.

To deactivate an AcceptFile function, call SetAcceptFileFunc with NoAcceptFile as the AcceptFile function.

Example

```
SetAcceptFileFunc(AcceptOneFile);
```

Enable the built-in AcceptFile function AcceptOneFile.

See Also

GetPathname

SetOverwriteOption

SetReceiveFilename

Syntax

```
procedure SetActualBPS(P : ProtocolRecPtr; BPS : LongInt);
procedure AbstractProtocol.SetActualBPS(BPS : LongInt);
```

Purpose

Set the actual BPS rate (needed only if modem differs from port).

Description

This routine can be used in cases like the following:

- Two machines are communicating via MNP or V.32 modems at 9600 BPS, and
- Both modems are using their built-in data compression facilities to increase the *effective* BPS rate (perhaps to 11000 BPS), and
- The machines are communicating with their modems at a rate of 19200 baud, to insure that the modem doesn't have to waste time waiting for data to send, and they are using hardware flow control to pace the flow of data between the modems and the machines

In such a case, the protocol will base any transfer rate calculations on a BPS rate of 19200, the rate at which data is being sent to the modem, and consequently it will underestimate the efficiency of the transmission, since the data isn't actually being sent to the remote at that rate.

To get more accurate calculations in this case, call SetActualBPS with a BPS parameter of 9600, the rate at which data is being sent to the remote's modem. The calculations will still be slightly inaccurate, because they will not account for the effects of the data compression being done by the modem. But the results will better reflect the actual efficiency of the transfer.

Example

```
var
  UP : UartPort;
  YP : YmodemProtocol;
  CM : CourierModem;
...
UP.InitCustom(Com1, 19200, NoParity, 8, 1, 2048, 2048, DefPortOptions);
UP.HWFlowEnable(1800, 200, hfUseRTS+hfRequireCTS);
UP.SetDTR(True);
...
CM.Init(@UP);
CM.SetFlowControl(fHWTrans+fHWRec);
CM.SetDataCompression(True);
...
YP.Init(@UP, True, False);
YP.SetActualBPS(9600);
```

The port is set up for 19200 baud and hardware flow control is enabled. The modem, a Courier HST, is configured to use hardware flow control and data compression. Since the actual modem-to-modem data transfer rate will be 9600 BPS, SetActualBPS is called to tell the protocol to use that value when making calculations concerning the efficiency of the transmission.

See Also

AbstractModem.SetDataCompression

SetBackgroundProc / SetDestinationDirectory

Syntax

```
procedure SetBackgroundProc(P : ProtocolRecPtr; BP : UserBackProc);  
procedure AbstractProtocol.SetBackgroundProc(BP : UserBackProc);
```

Purpose

Set a background procedure to be called while a file is transferred.

Description

Use this procedure to set a background hook to be called by ProtocolTransmit and ProtocolReceive whenever the protocol is in a psWaiting state.

To deactivate a user background procedure, call SetBackgroundProc with NoUserBack as the background procedure.

Examples

See "The User Background Hook" earlier in this section for more information and examples.

Syntax

```
procedure SetDestinationDirectory(P : ProtocolRecPtr; Dir : DirStr);  
procedure AbstractProtocol.SetDestinationDirectory(Dir : DirStr);
```

Purpose

Set the destination directory for received files.

Description

This routine allows you to specify the drive:\directory (Dir) where received files are to be stored. If only a drive is specified (e.g., 'D:'), the files are stored in the current directory for that drive.

Example

```
SetDestinationDirectory(P, 'C:\TEMP');
```

Store received files in C:\TEMP.

See Also

SetReceiveFilename

Syntax

```
procedure SetEfficiencyParms(P : ProtocolRecPtr;
                             BlockOverhead, TurnAroundDelay : Word);
procedure AbstractProtocol.SetEfficiencyParms(BlockOverhead,
                                              TurnAroundDelay : Word);
```

Purpose

Set the efficiency parameters for EstimateTransferSecs.

Description

This routine can be used to modify two of the parameters used in the Async Professional calculations of transfer time estimates. When you call EstimateTransferSecs, Async Professional employs the following logic:

```
ActualCPS      = connect speed
TurnDelay      = time (in milliseconds) for the block to get to the remote,
                  for the remote to acknowledge, and for the acknowledgement
                  to be returned
Overhead       = number of overhead bytes per block
BlockLen       = number of data bytes per block
Efficiency     = ratio of data bytes to highest possible number of bytes,
                  calculated as follows:
                  BlockLen
                  -----
                  BlockLen + Overhead + ((TurnDelay * ActualCPS) div 100)
EffectiveCPS = highest possible CPS accounting for Overhead, TurnDelay, and
                  ActualCPS, calculated as follows:
                  ActualCPS * Efficiency
```

EstimateTransferSecs uses EffectiveCPS to calculate time required to transfer a given number of bytes. If EffectiveCPS is wrong, the time estimates will be wrong. Since ActualCPS and BlockLen are determined by the connection and the protocol, the two variables under your control are Overhead and TurnDelay. Async Professional attempts to provide reasonable estimates for these values which in most cases should prove acceptable. However, should time estimates be consistently high or consistently low for your environments, you might want to consider adjusting TurnDelay and Overhead.

Example

APXMODEM and OOXMODEM, like all protocols, define default values for Overhead and TurnDelay:

```
const
  XmodemOverhead : Word = 5;      {Overhead bytes for each block}
  XmodemTurnDelay : Word = 1000;  {MSec turnaround delay for each block}
```

For two directly connected 386 machines, TurnDelay might be modified as follows:

```
var P : ProtocolRecPtr;
...
  InitXmodem(P, ComPort);
  SetEfficiencyParms(P, XmodemOverhead, 500);
...
```

The number of overhead bytes remains at the default, but the TurnDelay is estimated to drop by 2X.

See Also

EstimateTransferSecs

SetFileList

Syntax

```
procedure SetFileList(P : ProtocolRecPtr; FLP : FileListPtr);  
procedure AbstractProtocol.SetFileList(FLP : FileListPtr);
```

Purpose

Set the file list to use for the built-in NextFileList function.

Description

This routine stores the address of the file list pointed to by FLP for use by the supplied NextFile function NextFileList.

Example

```
var  
  P : ProtocolRecPtr;  
  FLP : FileListPtr;  
...  
  {...initialize P...}  
  MakeFileList(P, FLP, 1000);  
  {...check for error...}  
  AddFileToList(P, FLP, 'READ.ME');  
  AddFileToList(P, FLP, 'README.TOO');  
  {...add other files to list, checking for errors along the way...}  
  SetFileList(P, FLP);  
  SetNextFileFunc(P, NextFileList);  
  ProtocolTransmitZM(P);  
  DisposeFileList(P, FLP, 1000);
```

Outlines the basic technique needed to create a file list to be used by NextFileList. When using objects, the same routines would be called, but the initial P parameter would be omitted in each case.

See Also

AddFileToList
DisposeFileList

MakeFileList
SetNextFileFunc

Syntax

```
procedure SetFileMask(P : ProtocolRecPtr; NewMask : PathStr);  
procedure AbstractProtocol.SetFileMask(NewMask : PathStr);
```

Purpose

Set the directory/file mask for the built-in NextFileMask function.

Description

This routine sets the file mask to be used by the supplied NextFile function NextFileMask. Only one file mask can be used per transfer.

Example

```
SetNextFileFunc(P, NextFileMask);  
SetFileMask(P, 'C:\UPLOAD\*.ZIP');
```

This example shows how to tell NextFileMask to transfer all files with an extension of ZIP in the C:\UPLOAD directory. When using objects, the P parameters would be omitted.

See Also

SetNextFileFunc

Syntax

```
procedure SetHandshakeWait(P : ProtocolRecPtr; NewHandShake, NewRetry : Word);  
procedure AbstractProtocol.SetHandshakeWait(NewHandshake, NewRetry : Word);
```

Purpose

Set the wait times for the initial handshaking.

Description

This routine allows you to change the way that the protocol performs handshaking when it tries to initiate a file transfer. In most cases, the protocol will retry up to 10 times (DefHandshakeRetry), waiting up to 10 seconds (DefHandshakeWait = 182 clock ticks) for a response on each try. SetHandshakeWait allows you to change these values. NewHandshake is the amount of time (in clock ticks) to wait on each try. NewRetry is the maximum number of times to retry.

Note that SetHandshakeWait does not pertain to the ASCII protocol, which does not perform handshaking, and that the default values are different for the Zmodem protocol, which normally waits up to 60 seconds (1092 clock ticks) on each try.

Example

```
SetHandshakeWait(546, 10);
```

Configure protocol to wait up to 30 seconds ($30 \times 18.2 = 546$) on each try.

See Also

SetBlockWait (APXMODEM)

XmodemProtocol.SetBlockWait

SetLogFileProc

Syntax

```
procedure SetLogFileProc(P : ProtocolRecPtr; LFP : LogFileProc);  
procedure AbstractProtocol.SetLogFileProc(LFP : LogFileProc);
```

Purpose

Set a procedure to log file transfers.

Description

This routine allows you to specify a procedure (LFP) that is called at various stages during a file's transmission, giving you an opportunity to create a log of events. This hook is particularly useful when writing BBS software, but it is useful in other applications and for debugging purposes.

If you are using APABSPCL, a LogFile procedure should be of the following form:

```
{ $F+ }  
procedure LogFile(P : ProtocolRecPtr; LogFileStatus : LogFileType);  
begin  
    ...  
end;
```

P is a pointer to the data used to process the protocol. LogFileStatus indicates the current event. Possible values for LogFileStatus are:

lfReceiveStart	- a file is about to be received
lfReceiveOk	- the file was received successfully
lfReceiveFail	- the receive operation failed
lfReceiveSkip	- the file was rejected by the protocol for file management reasons
lfTransmitStart	- a file is about to be sent
lfTransmitOk	- the file was sent successfully
lfTransmitFail	- the file transfer failed
lfTransmitSkip	- the file was rejected by the protocol for file management reasons

lfReceiveSkip and lfTransmitSkip are used only when the file fails Zmodem's file management rules or when your AcceptFile function returns False under any protocol.

See "Logging Procedure" earlier in this section for more information.

If using OOABSPCL:

```
{ $F+ }  
procedure LogFile(AP : AbstractProtocolPtr; LogFileStatus : LogFileType);  
begin  
    ...  
end;
```

There are no built-in LogFile procedures other than the default one, NoLogFile, which does nothing.

To deactivate a LogFile procedure, call SetLogFileProc with NoLogFile as the LogFile procedure.

Example

For an example of a LogFile routine, see the LogFileActivity procedures in FX.PAS and FXO.PAS. See also the example programs EXXFR1.PAS and EXXFRO1.PAS.

Syntax

```
procedure SetNextFileFunc(P : ProtocolRecPtr; NFFunc : NextFileFunc);  
procedure AbstractProtocol.SetNextFileFunc(NFFunc : NextFileFunc);
```

Purpose

Set a function for batch protocols to call to get a file to transmit.

Description

This routine can be used in connection with protocols that support batch file transfers to designate a function (NFFunc) that will return the name of the next file to transmit.

If you are using APABSPCL, a NextFile function should be of the following form:

```
{F+}  
function NextFile(P : ProtocolRecPtr; var FName : PathStr) : Boolean;  
begin  
    NextFile := True;  
    FName := {next file to transmit} ;  
end;
```

P is a pointer to the data used to process the protocol. FName is the name of the next file to transmit. The function should return False if there are no more files to transmit, otherwise True.

If using OOABSPCL:

```
{F+}  
function NextFile(AP : AbstractProtocolPtr;  
                  var FName : PathStr) : Boolean;  
begin  
    NextFile := True;  
    FName := {next file to transmit} ;  
end;
```

Both APABSPCL and OOABSPCL provide two built-in NextFile functions. The default function, NextFileMask, transmits all files matching the file mask specified by a call to SetFileMask. The other function, NextFileList, transmits all files in the list specified by a call to SetFileList.

See "NextFile Function" earlier in this section for more information.

Example

See the examples for SetFileList and SetFileMask.

See Also

SetFileList

SetFileMask

SetOverwriteOption / SetProtocolPort

Syntax

```
procedure SetOverwriteOption(P : ProtocolRecPtr; Opt : WriteFailOptions);  
procedure AbstractProtocol.SetOverwriteOption(Opt : WriteFailOptions);
```

Purpose

Set option for what to do when the destination file already exists.

Description

This routine sets the behavior of the protocol in cases where a file being received already exists. If Opt is WriteFail (the default), the protocol aborts the transfer if the file already exists. If Opt is WriteRename, the first character of the filename is changed to '\$' (e.g., 'SAMPLE.DOC' becomes '\$AMPLE.DOC'). If a file already exists with the new '\$' filename, it is overwritten without warning.

If Opt is WriteAnyway, the existing file is simply overwritten. The code in Async Professional that handles this option is executed after the AcceptFile function has been called. If your AcceptFile function already deals with the issue of potential overwrites of existing files, call SetOverwriteOption with a parameter of WriteAnyway.

These options *do not* apply to Zmodem protocols since the Zmodem protocol applies its own file management rules.

Example

```
SetAcceptFileFunc(P, MyAcceptFile);  
SetOverwriteOption(P, WriteAnyway);
```

Enable MyAcceptFile, which deals with the existing file issue, and set the overwrite option to WriteAnyway.

See Also

SetAcceptFileFunc

Syntax

```
procedure SetProtocolPort(P : ProtocolRecPtr; PortPtr : PortRecPtr);  
procedure AbstractProtocol.SetProtocolPort(AP : AbstractPortPtr);
```

Purpose

Set the port for this protocol.

Description

This routine tells a protocol to use the port specified by PortPtr. It is especially important to call this method if, for example, you are using objects and the port object originally passed to the protocol object's constructor was deallocated and then reallocated. You cannot assume that the object will be allocated at the same place on the heap the second time.

Example

```
var  
  UPP1, UPP2 : UartPortPtr;  
  XP : XmodemProtocol;  
...  
  UPP1 := New(UartPortPtr, InitFast(Com1, 2400));  
  UPP2 := New(UartPortPtr, InitFast(Com2, 2400));  
  XP.Init(UPP1, False, False);  
...  
  XP.SetProtocolPort(UPP2);
```

Initialize XP for use with Com1; later switch to Com2.

Syntax

```
procedure SetReceiveFilename(P : ProtocolRecPtr; Fname : PathStr);  
procedure AbstractProtocol.SetReceiveFilename(Fname : PathStr);
```

Purpose

Give a name to the file to be received.

Description

This routine allows you to specify the name of a file about to be received. There are only two circumstances where this routine is needed. The first is when using a protocol that does not send the name of the file before it starts sending data. The only such protocols that are currently supported are ASCII and Xmodem.

The second case is inside an AcceptFile function where you want to change the name of the file. In fact, this is the *only* time you can use this function for protocols that provide the name of the received file. Used at any other time, the name you specify will be ignored.

Example

```
SetReceiveFilename(P, 'C:\DOWNLOAD\RECEIVE.TMP');  
Store the file about to be received in C:\DOWNLOAD\RECEIVE.TMP.
```

Syntax

```
procedure SetShowStatusProc(P : ProtocolRecPtr; SProc : ShowStatusProc);  
procedure AbstractProtocol.SetShowStatusProc(SProc : ShowStatusProc);
```

Purpose

Set a user status function.

Description

This routine allows you to specify a routine that can be called during a file transfer to display status information (amount of data sent/received, time remaining, etc.).

If you are using APABSPCL, a ShowStatus routine should be of the following form:

```
{SF+}  
procedure ShowStatus(P : ProtocolRecPtr; Starting, Ending : Boolean);  
begin  
  ...  
end;
```

P is a pointer to the data used to process the protocol. It can be passed to the various GetXxxx routines to obtain status information. Starting is True when ShowStatus is called the first time, giving you a chance to initialize any variables needed by ShowStatus or to allocate space on the heap as needed. Ending is True when ShowStatus is called for the last time by the protocol, giving you a chance to dispose of any heap space that was allocated.

If using OOABSPCL:

```
{SF+}  
procedure ShowStatus(AP : AbstractProtocolPtr;  
                    Starting, Ending : Boolean);  
begin  
  ...  
end;
```

See "Status Procedure" earlier in this section for more information.

To deactivate a ShowStatus function, call SetShowStatusFunc with NoStatus as the ShowStatus function.

Example

See the example programs FX.PAS, FXO.PAS, EXXFR.PAS, and EXXFRO.PAS.

SupportsBatch

Syntax

```
function SupportsBatch(P : ProtocolRecPtr) : Boolean;  
function AbstractProtocol.SupportsBatch : Boolean;
```

Purpose

Return True if this protocol supports batch file transfers.

Description

This function returns True if the protocol currently in use permits batch file transfers (Kermit, Ymodem, and Zmodem do; Xmodem, B+, and ASCII do not). The most common use for it is when the user is being asked for the name of a file to transfer. If the protocol supports batch file transfers, it is generally acceptable for the filename to include DOS wildcard characters; if the protocol doesn't, it isn't.

Example

```
Write('File to upload: ');  
ReadLn(S);  
if not SupportsBatch(P) then  
  if (Pos('*', S) <> 0) or (Pos('?', S) <> 0) then begin  
    WriteLn('^G, 'DOS wildcard characters not allowed!');  
    Exit;  
  end;
```

Prompt the user for the name of a file to send. If batch file transfers are not possible, disallow DOS wildcard characters in the name.

See Also

GetProtocol

D. Xmodem Protocol: APXMODEM/OOXMODEM

APXMODEM/OOXMODEM build on the services of APABSPCL/OOABSPCL to provide the ubiquitous Xmodem protocol. Xmodem is the oldest protocol supported by Async Professional. It was developed and first implemented by Ward Christensen in 1977 and placed in the public domain. Since then, it's become an extremely popular protocol and continues in use today (though at a diminished frequency).

Xmodem is also the simplest, and perhaps the slowest, protocol supported by Async Professional. Xmodem uses blocks of only 128 bytes, requires an acknowledgement of each block, and uses only a simple checksum for data integrity.

What follows is a simplified discussion of the Xmodem protocol, although it describes far more than is required to actually use the protocol in Async Professional. For additional details on the internals of the Xmodem protocol, see the YMODEM.DOC file in the PROTDOL.ZH archive.

Xmodem blocks have the following format:

<SOH>	<block#>	not <block#>	<128 bytes of data>	<checksum>
-------	----------	-----------------	---------------------	------------

The <SOH> character marks the start of the block. Next comes a one byte block number followed by a one's complement of the block number. The block number starts at one and goes up to 255 where, being a byte field, it rolls over to zero and starts the cycle again. Following the block numbers are 128 bytes of data and a one-byte checksum. The checksum is calculated by adding together all the data bytes and ignoring any carries that might result.

A typical Xmodem protocol transfer looks something like this:

Transmitter		Receiver
	<---	<NAK>
<SOH><1><254><128 data bytes><chk>	---->	
	<---	<ACK>
<SOH><2><253><128 data bytes><chk>	---->	
	<---	<ACK>
<EOT>	---->	
	<---	<ACK>

The receiver always starts the protocol by issuing a <NAK>, also called the handshake character. It waits 10 seconds for the transmitter to send a block. If it doesn't get a block within 10 seconds, it sends another <NAK>. This continues for 10 retries; then the receiver gives up.

If the receiver does get a block, it compares the checksum it calculates to the received checksum. If they differ, the receiver sends a <NAK> and the transmitter resends the block. If the checksums are the same, the receiver accepts the block by sending an <ACK>. This continues until the complete file is transmitted. The transmitter signifies this by sending an <EOT>.

Either side can cancel the protocol at any time by sending 3 <CAN> characters (^X).

From this description several things become clear. First, this protocol does not include any information about the file being transmitted. Therefore, the receiving program, or the person using the receiving program, must assign a name to the file being received.

The receiving program also doesn't know the exact size of the file. Hence, the received file size is always a multiple of the block size. The Async Professional Xmodem protocol fills transmitted blocks with BlockFillChar characters (the default is '^Z').

Xmodem often exhibits a startup delay. The transmitter always waits for a <NAK> from the receiver as its start signal. If the receiving program was started first, the transmitter probably missed the first <NAK>. It must wait for the receiver to timeout and send another <NAK> (usually 10 seconds).

Also, Xmodem offers no "escaping" of binary control characters. "Escaping" means that characters may be transformed before being transmitted. This is done to prevent certain binary data characters, such as <Xon>, from being interpreted as flow control characters. Since Xmodem doesn't escape any binary characters, you can't use software flow control in an Xmodem transfer (since the flow control software would misinterpret <Xon> or <Xoff> characters in the data stream as a flow control request).

In summary, the *only* merit of the basic Xmodem protocol is that it's so widespread it's probably supported by any microcomputer communications program you might find, thus providing a lowest common denominator between systems.

Xmodem Extensions

Xmodem has been tweaked and improved through the years. Some of these tweaks have become standards of their own and are supported by Async Professional. These Xmodem extensions are enabled by turning on various options in the Xmodem object or record.

The first of these improvements is called Xmodem CRC. It substitutes a 16 bit CRC (cyclic redundancy check) for the original checksum. This offers a *much* higher level of data integrity. When given the opportunity, you should always choose Xmodem CRC over plain Xmodem. In fact, Async Professional always assumes that you want to use Xmodem CRC and only drops back to using checksums if the remote computer can't use CRCs.

The receiver indicates that it wants to use Xmodem CRC by sending the character 'C' instead of <NAK> to start the protocol. If the transmitter doesn't respond to the 'C' within three attempts, the receiver assumes the transmitter isn't capable of using Xmodem CRC. The receiver automatically drops back to using checksums by sending a <NAK>.

Another popular extension is called Xmodem 1K. This derivative builds on Xmodem CRC by using 1024 byte blocks instead of 128 byte blocks. When using 1024 byte blocks, each block starts with an <STX> character rather than an <SOH>. Xmodem 1K also uses a 16 bit CRC as the block check.

Using a larger block size can greatly speed up the protocol because it reduces the number of times the transmitter must wait for an acknowledgement. However, it might actually *reduce* throughput over bad lines because more data must be retransmitted when errors are encountered. Async Professional Xmodem 1K drops back to 128 byte blocks whenever it receives more than 5 <NAK>s in a row. Once it drops back to 128 byte blocks, it never tries to step back up to 1024 byte blocks.

To request Xmodem 1K, simply pass True for the Use1K parameter when initializing the protocol; or call Set1KMode(True) after initializing the protocol.

The final Xmodem extension supported by Async Professional is Xmodem 1KG. This is a "streaming" protocol and is requested when the receiver sends 'G' as the initial handshake character instead of <NAK>. Streaming, in this context, means that the transmitter continuously transmits

blocks without waiting for acknowledgements. In fact, the receiver *never* sends an acknowledgement. All blocks are assumed to be correct. If the receiver does encounter a bad block, it aborts the entire transfer by sending a <NAK>.

Obviously, you wouldn't even consider using this streaming protocol unless you are using error correcting modems and both modems have their error control features turned on. In fact, you might want to have your application refuse to use Xmodem 1KG if error correcting modems aren't detected (see `GetLastErrorMode` in §6.C for information on how to detect error correcting connections).

The advantage of a streaming protocol like Xmodem 1KG is its very high throughput. There is *no* turnaround delay, so the protocol is extremely efficient. To request Xmodem 1KG, just pass `True` for the `UseGMode` parameter when initializing the Xmodem protocol; or call `SetGMode(True)` after initializing the protocol.

Shared Declarations

Constants

`DefBlockWait = 91;`

Default length of time (in clock ticks) to wait between blocks for a response from the remote (91 clock ticks = 5 seconds). See the entry for `SetBlockWait`.

`DefFinishWait : Word = DefBlockWait*2;`

Default length of time (in clock ticks) to wait after an <EOT> for a response from the remote.

`DefMaxBlockErrors : Byte = 5`

Default number of errors that can occur for a single block before the file transfer is aborted automatically.

`RelaxedBlockWait = 182;`

This parameter is passed to `SetBlockWait` when "relaxed Xmodem" is desired (182 clock ticks = 10 seconds).

`RelaxedHandshakeWait = 364;`

This parameter is passed to `SetHandshakeWait` (APABSPCL/OOABSPCL) when "relaxed Xmodem" is desired (364 clock ticks = 20 seconds).

`XmodemOverhead : Word = 5;`

When estimating transfer times, `EstimateTransferSecs` (APABSPCL/OOABSPCL) assumes that there are 5 overhead bytes for each data block. See `SetEfficiencyParms` (APABSPCL/OOABSPCL) for more information.

`XmodemTurnDelay : Word = 1000;`

When estimating transfer times, `EstimateTransferSecs` (APABSPCL/OOABSPCL) assumes that there will be a 1000 millisecond (1 second) delay from the time it sends a block until it receives an acknowledgement. See `SetEfficiencyParms` (APABSPCL/OOABSPCL) for more information.

APXMODEM Declarations

Types

`XmodemPtr = ^XmodemProtocol;`

`XmodemProtocol =`

`record`

`...`

`end;`

Record used to store data needed by the various routines in APXMODEM to implement the Xmodem protocol and its variants (relaxed Xmodem, Xmodem CRC, Xmodem 1K, and Xmodem 1KG).

Procedures and Functions

```
procedure PrepareReceivePartXM(P : ProtocolRecPtr);
  {-Prepare to call the background function ProtocolReceivePartXM}
procedure PrepareTransmitPartXM(P : ProtocolRecPtr);
  {-Prepare to use the background function ProtocolTransmitPartXM}
procedure ProtocolReceiveXM(P : ProtocolRecPtr);
  {-Start Xmodem protocol receive}
function ProtocolReceivePartXM(P : ProtocolRecPtr) : ProtocolStateType;
  {-Perform one increment of a background protocol receive}
procedure ProtocolTransmitXM(P : ProtocolRecPtr);
  {-Start Xmodem protocol transmit}
function ProtocolTransmitPartXM(P : ProtocolRecPtr) : ProtocolStateType;
  {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the similarly named functions in APABSPCL (for example, PrepareReceivePartXM is documented in PrepareReceivePart). See §7.C for a complete discussion of these functions.

OOXMODEM Declarations

Types

```
XmodemProtocolPtr = ^XmodemProtocol;
XmodemProtocol =
  object(AbstractProtocol)
  ...
end;
```

Object that defines data fields and methods needed to implement the Xmodem protocol and its variants (relaxed Xmodem, Xmodem CRC, Xmodem 1K, and Xmodem 1KG).

Procedures and Functions

```
procedure XmodemProtocol.PrepareReceivePart; virtual;
  {-Prepare to call the background function ProtocolReceivePart}
procedure XmodemProtocol.PrepareTransmitPart; virtual;
  {-Prepare to use the background function ProtocolTransmitPart}
procedure XmodemProtocol.ProtocolReceive; virtual;
  {-Start Xmodem protocol receive}
function XmodemProtocol.ProtocolReceivePart : ProtocolStateType ; virtual;
  {-Perform one increment of a background protocol receive}
procedure XmodemProtocol.ProtocolTransmit; virtual;
  {-Start Xmodem protocol transmit}
function XmodemProtocol.ProtocolTransmitPart : ProtocolStateType ; virtual;
  {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the ones with the same name in OOABSPCL. See §7.C for a complete discussion.

Syntax

```
procedure DoneXmodem(var P : ProtocolRecPtr);
```

Purpose

Dispose of a protocol record.

Description

This routine disposes of the protocol record pointed to by P, previously allocated by a call to InitXmodem or InitCustomXmodem.

Example

See the examples for InitCustomXmodem and InitXmodem.

See Also

InitXmodem

Syntax

```
constructor XmodemProtocol.Init(APPtr : AbstractPortPtr;  
                                Use1K, UseGMode : Boolean);
```

Purpose

Allocate and initialize a protocol control block.

Description

This constructor initializes an XmodemProtocol object using the default protocol options (DefProtocolOptions). See InitCustom for details.

Example

```
var  
    UP : UartPort;  
    XP : XmodemProtocol;  
...  
    UP.InitFast(Com1, 2400);  
...  
    if not XP.Init(@UP, False, False) then  
        {handle error} ;  
...  
    XP.Done;
```

Set up for an Xmodem file transfer at 2400 baud using Com1.

See Also

XmodemProtocol.InitCustom

InitXmodem

InitCustom

Syntax

```
constructor XmodemProtocol.InitCustom(APPtr : AbstractPortPtr;  
                                     Use1K, UseGMode : Boolean;  
                                     Options : Word);
```

Purpose

Allocate and initialize a protocol control block with the specified options.

Description

This constructor instantiates an object of type XmodemProtocol in order to transfer files using the Xmodem protocol. The APPtr parameter specifies the port to be used. If Use1K is True, the Xmodem 1K protocol is used, rather than regular Xmodem or Xmodem CRC. If UseGMode is True, the Xmodem 1KG protocol is used instead. The Options parameter indicates which protocol options are in effect. Normally it is DefProtocolOptions (see APABSPCL/OPABSPCL in §7.C), in which case you can use the Init constructor instead of InitCustom.

Since the receiver specifies the block check type (checksum or CRC), the Xmodem object transmits with whatever check type the receiver specifies. When the Xmodem object is the receiver, it *always* attempts to use CRC block checking. If the remote Xmodem doesn't support CRC, then this object switches to using checksums.

If you want to use checksums initially (not recommended), set the CheckType field of the protocol object directly, as follows:

```
XP.CheckType := bcChecksum1;
```

Example

```
var  
  UP : UartPort;  
  XP : XmodemProtocol;  
...  
  UP.InitFast(Com1, 2400);  
  ...  
  if not XP.InitCustom(@UP, False, False, DefProtocolOptions) then  
    {handle error} ;  
  ...  
  XP.Done;
```

Set up for an Xmodem file transfer at 2400 baud using Com1.

See Also

XmodemProtocol.Init

InitCustomXmodem

Syntax

```
procedure InitCustomXmodem(var P : ProtocolRecPtr; PortPtr : PortRecPtr;  
                          Use1K, UseGMode : Boolean; Options : Word);
```

Purpose

Allocate and initialize a protocol control block with the specified options.

Description

This routine is used to initialize a protocol record, to be used to transfer files under the Xmodem protocol. The P parameter specifies the port to be used. If Use1K is True, the Xmodem 1K protocol is used, rather than regular Xmodem or Xmodem CRC. If UseGMode is True, the Xmodem 1KG protocol is used instead. The Options parameter indicates which protocol options are in effect. Normally it is DefProtocolOptions (see APABSPCL/OPABSPCL in §7.C), in which case you can use InitXmodem instead of InitCustomXmodem.

Example

```
var XP : ProtocolRecPtr;  
    P : PortRecPtr;  
  
...  
    InitPortFast(P, Com1, 2400);  
    ...  
    InitCustomXmodem(XP, P, False, False, DefProtocolOptions);  
    ...  
    DoneXmodem(XP);
```

Set up for an Xmodem file transfer at 2400 baud using Com1.

See Also

XmodemProtocol.InitCustom

InitXmodem

Syntax

```
procedure InitXmodem(var P : ProtocolRecPtr; PortPtr : PortRecPtr;  
                   Use1K, UseGMode : Boolean);
```

Purpose

Allocate and initialize a protocol control block.

Description

This routine initializes a protocol record for Xmodem file transfers using the default protocol options (DefProtocolOptions). See InitCustomXmodem for details.

Example

```
var XP : ProtocolRecPtr;  
    P : PortRecPtr;  
  
...  
    InitPortFast(P, Com1, 2400);  
    ...  
    InitXmodem(XP, P, False, False);  
    ...  
    DoneXmodem(XP);
```

Set up for an Xmodem file transfer at 2400 baud using Com1.

Load

Syntax

```
{IFDEF UseStreams}  
constructor XmodemProtocol.Load(var S : IdStream);
```

Purpose

Load an XmodemProtocol object from a stream.

Description

This constructor loads an XmodemProtocol object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to XmodemProtocol.Store.

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Load is called.

See the entry for XmodemProtocol.Store for a discussion of how to link a protocol object loaded from a stream with a port object.

The stream registration routine for an XmodemProtocol object is XmodemProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses  
  OpRoot, ApPort, OoCom, OoAbsPcl, OoXmodem;  
var  
  UP : UartPort;  
  XP : XmodemProtocol;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not S.Init('XMODEM.STM', SOpen, 1024) then begin  
    WriteLn('Error opening stream');  
    Halt;  
  end;  
  S.RegisterHier(XmodemProtocolStream);  
  S.RegisterHier(UartPortStream);  
  S.RegisterPointer(1000, @UP);  
  S.Get(UP);  
  S.Get(XP);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Load error: ', Status);  
    Halt;  
  end;  
  WriteLn('Loaded successfully');  
  S.Done;  
  XP.Done;  
  UP.Done;  
end.
```

Instantiates UP and XP from a stream created by the Store example using the stream's Get method, which calls the objects' Load constructors indirectly.

Notice that a pointer to the port object is registered to ensure that XmodemProtocol.Load can create a link between the two objects.

Syntax

```
procedure Set1KMode(P : ProtocolRecPtr; Enable : Boolean);  
procedure XmodemProtocol.Set1KMode(Enable : Boolean);
```

Purpose

Enable/disable Xmodem 1K.

Description

This routine is used to select Xmodem 1K after a protocol record/object has already been initialized.

Example

```
InitXmodem(XP, P, False, False);  
...  
Set1KMode(XP, True);  
Configure XP for regular Xmodem initially; switch later to Xmodem 1K.
```

See Also

XmodemProtocol.InitCustom	SetGMode
InitCustomXmodem	

Syntax

```
procedure SetBlockWait(P : ProtocolRecPtr; NewBlockWait : Byte);  
procedure XmodemProtocol.SetBlockWait(NewBlockWait : Byte);
```

Purpose

Set the interblock wait time.

Description

This routine is intended to be used when transmitting files to or from a BBS or information service (such as CompuServe) that tends to delay too long between blocks. By default, Xmodem will wait up to 5 seconds (91 clock ticks) between blocks. SetBlockWait allows you to increase the length of the waiting period to prevent needless errors caused by delays at the remote's end of the connection.

The typical NewBlockWait parameter is RelaxedBlockWait (= 182 clock ticks), which specifies a wait time of 10 seconds.

Example

```
InitXmodem(XP, P, False, False);  
SetHandshakeWait(XP, RelaxedHandshakeWait, 10);  
SetBlockWait(XP, RelaxedBlockWait);  
Set up for a file transfer using "relaxed Xmodem."
```

See Also

SetHandshakeWait (APABSPCL)	AbstractProtocol.SetHandshakeWait
-----------------------------	-----------------------------------

SetFinishWait / SetGMode

Syntax

```
procedure SetFinishWaitXM(P : ProtocolRecPtr; NewFinishWait : Word);  
procedure XmodemProtocol.SetFinishWait(NewFinishWait : Word);
```

Purpose

Change the finish wait time (time to wait for the response to an <EOT>).

Description

When using large transmit buffers, MNP modems, or noisy links, it is possible for the transmitter to timeout while waiting for a response to its <EOT> block. Async Professional attempts to avoid this problem by not sending an <EOT> until the output buffer is drained. It also uses a default wait time of 10 seconds (twice the default block wait time). In most cases this is sufficient to avoid <EOT> response timeouts. However, if you do encounter <EOT> response timeouts, use this function to increase the <EOT> wait time. This gives the protocol some extra time to wait for the <EOT> response. NewFinishWait is the new wait time, in ticks, to wait for a response to the <EOT>.

Example

```
XP^.SetFinishWait(Secs2Tics(15));
```

Changes the wait time to 15 seconds.

Syntax

```
procedure SetGMode(P : ProtocolRecPtr; Enable : Boolean);  
procedure XmodemProtocol.SetGMode(Enable : Boolean);
```

Purpose

Enable or disable streaming (Xmodem 1KG).

Description

This routine can be used to select Xmodem 1KG after a protocol record/object is initialized. Note that it is not necessary to call Set1KMode if Enable is True, since SetGMode does so automatically, but you do need to call Set1KMode if Enable is False and you want to switch to regular Xmodem rather than Xmodem 1K.

Example

```
InitXmodem(XP, P, False, False);  
...  
SetGMode(XP, True);  
...  
SetGMode(XP, False);  
Set1KMode(XP, False);
```

Configure XP for regular Xmodem initially; switch to Xmodem 1KG; switch back to regular Xmodem.

See Also

XmodemProtocol.InitCustom
InitCustomXmodem

Set1KMode

Syntax

```
($IFDEF UseStreams)
procedure XmodemProtocol.Store(var S : IdStream);
```

Purpose

Store an XmodemProtocol object to a stream.

Description

Store writes an image of the XmodemProtocol object to the stream S.

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Store is called.

More importantly, you must give XmodemProtocol.Store some means of forming a link between the protocol object and the port object whose address was passed to the Init constructor. There are three ways to do so:

1. Store the port object in the stream and register a pointer to it prior to storing the protocol object. (See the example below.) When reloading from the stream, you must then load the port object and register a pointer to it prior to loading the protocol object. (See the example for XmodemProtocol.Load.)
2. Don't store the port object in the stream, but register a pointer to it before storing the protocol object. When reloading from the stream, register a pointer to the port object to be used before loading the protocol object.
3. Don't store the port object and don't register a pointer to it, but call the corresponding stream registration routine (e.g., execute S.RegisterHier(UartPortStream);). XmodemProtocol.Store automatically stores the port object in the stream. When the protocol object is reloaded, the port object is loaded automatically by XmodemProtocol.Load. You can then obtain a pointer to it by accessing the protocol object's APort data field (of type AbstractPortPtr). In this case, XmodemProtocol.Done will automatically dispose of the port object that was loaded from the stream.

Of the three, the second scenario is probably the most common.

The stream registration routine for an XmodemProtocol object is XmodemProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses
  OpRoot, ApPort, OoCom, OoAbsPcl, OoXmodem;
var
  UP : UartPort;
  XP : XmodemProtocol;
  S : BufIdStream;
  Status : Word;
begin
  if not UP.InitFast(Com1, 2400) then begin
    WriteLn('Error initializing port');
    Halt;
  end;
  if not XP.Init(@UP, False, False) then begin
    WriteLn('Error initializing protocol');
    Halt;
  end;
  if not S.Init('XMODEM.STM', SCreate, 1024) then begin
    WriteLn('Error initializing stream');
    Halt;
  end;
  S.RegisterHier(XmodemProtocolStream);
```

Store

```
S.RegisterHier(UartPortStream);
S.RegisterPointer(1000, @UP);
S.Put(UP);
S.Put(XP);
Status := S.GetStatus;
if Status <> 0 then begin
    WriteLn('Store error: ', Status);
    Halt;
end;
WriteLn('Stored successfully');
S.Done;
XP.Done;
UP.Done;
end.
```

Instantiates the port and protocol objects and then stores them in a stream using the stream's Put method, which calls the objects' Store methods indirectly.

Notice that a pointer to the UartPort object is registered so that XmodemProtocol.Store can create a link between the two objects.

See XmodemProtocol.Load for an example program that reloads these objects.

See Also

XmodemProtocol.Load

E. Ymodem Protocol: APYMODEM/OOYMODEM

The APYMODEM and OOYMODEM units build on APABSPCL/OOABSPCL and APXMODEM/OOXMODEM to provide the Ymodem protocol. Ymodem is, in every sense of the word, a derivative of Xmodem. The details and history of Ymodem are fully explained in the file YMODEM.DOC in the PROTDOLZ.H archive. What follows is a simplified explanation of Ymodem (which is more than enough to use it with Async Professional).

Ymodem is essentially Xmodem 1K with batch facilities added. A single Ymodem protocol can transfer as many files as you care to transmit. Additionally, and perhaps equally important, the transmitter can provide the receiver with information about the incoming file: the file name, actual size, and modification date.

Ymodem achieves this by adding a zero'th block to the Xmodem 1K protocol. Block zero is transferred first and contains file information in the following format:

<SOH>	<0>	<255>	<name><0><length><0><date><0>...<0>	<CRCh i><CRCl o>
-------	-----	-------	-------------------------------------	------------------

The <name> field is the only required field. It supplies the name of the file in lower case letters. Path information can be included but the protocol requires that all '\ ' characters be converted to '/' characters.

The <length> field specifies the file length. This allows the receiver to truncate the received file to discard the filler characters at the end of the last block.

The <date> field is the modification date of the file. This is the number of seconds since January 1, 1970 GMT, expressed in ASCII octal digits (a Unix convention).

Async Professional takes care of properly formatting this block; you don't need to do anything but specify the name of the file to transmit. When receiving Ymodem files, Async Professional uses the information in this block, if present, to adjust the size of the received file and its modification date.

As with the Xmodem protocols, the Ymodem protocol starts when the receiver sends its handshake character ('C') to the transmitter. The transmitter responds with a properly formatted block zero. The receiver acknowledges this with an <ACK> and then starts a normal Xmodem CRC protocol by issuing another 'C' handshake character.

Here's a sample Ymodem session:

Transmitter		Receiver
	<---	'C'
<SOH><0><255><file info><crc>	--->	
	<---	<ACK>
	<---	'C'
<STX><1><254><1024 data bytes><crc>	--->	
	<---	<ACK>
<STX><2><253><1024 data bytes><crc>	--->	
	<---	<ACK>
<EOT>	--->	
	<---	'C'
<SOH><0><255><file info><crc>	--->	
	<---	<ACK>
	<---	'C'
<STX><1><254><1024 data bytes><crc>	--->	
	<---	<ACK>

<EOT>	---	
<SOH><0><255><128 zeros><crc>	---	
	<---	<ACK>

Using Ymodem in Async Professional is as simple as instantiating a Ymodem object instead of an Xmodem object (or, in non-OOP format, calling InitYmodem rather than InitXmodem).

Ymodem Extensions

The Ymodem specification permits Ymodem to mix 128 and 1024 byte blocks. Most Ymodem protocols start with 1024 byte blocks and drop back to 128 byte blocks only if repeated errors are received (once the block size is reduced to 128 bytes, it is never stepped back up to 1024). With Async Professional, you specify the initial block size via the Use1K parameter in the Init and InitCustom constructors (or InitYmodem in non-OOP format). In almost all cases, you should start with 1024 byte blocks.

Like Xmodem, Ymodem also offers a "streaming" extension called Ymodem G. This is similar in performance (and drawbacks) to Xmodem 1KG, but offers the standard advantages of Ymodem (batch transfers and file information). Activate the streaming option for Ymodem by passing True for the UseGMode option. Or, call SetGMode(True) after initializing the protocol.

APYMODEM Declarations

Types

```
YmodemPtr = ^YmodemProtocol;
YmodemProtocol =
    record
    ...
    end;
```

Record used to store data needed by the various routines in APYMODEM to implement the Ymodem and Ymodem G protocols.

Procedures and Functions

```
procedure PrepareReceivePartYM(P : ProtocolRecPtr);
    {-Prepare to call the background function ProtocolReceivePartYM}
procedure PrepareTransmitPartYM(P : ProtocolRecPtr);
    {-Prepare to use the background function ProtocolTransmitPartYM}
procedure ProtocolReceiveYM(P : ProtocolRecPtr);
    {-Receive a batch of files transmitted using Ymodem}
function ProtocolReceivePartYM(P : ProtocolRecPtr) : ProtocolStateType;
    {-Perform one increment of a background protocol receive}
procedure ProtocolTransmitYM(P : ProtocolRecPtr);
    {-Start Ymodem protocol transmit}
function ProtocolTransmitPartYM(P : ProtocolRecPtr) : ProtocolStateType;
    {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the similarly named functions in APABSPCL (for example, PrepareReceivePartYM is documented in PrepareReceivePart). See §7.C for a complete discussion of these functions.

OOYMODEM Declarations

Types

```
YmodemProtocolPtr = ^YmodemProtocol;  
YmodemProtocol =  
    object(XmodemProtocol)  
    ...  
end;
```

Object that defines data fields and methods needed to implement the Ymodem and Ymodem G protocols.

Procedures and Functions

```
procedure YmodemProtocol.PrepareReceivePart; virtual;  
    {-Prepare to call the background function ProtocolReceivePart}  
procedure YmodemProtocol.PrepareTransmitPart; virtual;  
    {-Prepare to use the background function ProtocolTransmitPart}  
procedure YmodemProtocol.ProtocolReceive; virtual;  
    {-Receive a batch of files transmitted using Ymodem}  
function YmodemProtocol.ProtocolReceivePart : ProtocolStateType; virtual;  
    {-Perform one increment of a background protocol receive}  
procedure YmodemProtocol.ProtocolTransmit; virtual;  
    {-Start Ymodem protocol transmit}  
function YmodemProtocol.ProtocolTransmitPart : ProtocolStateType; virtual;  
    {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the ones with the same name in OOABSPCL. See §7.C for a complete discussion.

Done / DoneYmodem

Syntax

destructor YmodemProtocol.Done; virtual;

Purpose

Dispose of the protocol object.

Description

This Done method disposes of the internal data structures allocated on the heap by the Init method.

Example

See the example for InitCustom.

Syntax

procedure DoneYmodem(var P : ProtocolRecPtr);

Purpose

Dispose of a protocol record.

Description

This routine disposes of the protocol record pointed to by P, previously allocated by a call to InitYmodem or InitCustomYmodem.

Example

See the examples for InitYmodem and InitCustomYmodem.

See Also

InitYmodem

Syntax

```
constructor YmodemProtocol.Init(APPtr : AbstractPortPtr;  
                                Use1K, UseGMode : Boolean);
```

Purpose

Allocate and initialize a protocol control block.

Description

This constructor initializes a YmodemProtocol object using the default protocol options (DefProtocolOptions). See InitCustom for details.

Example

```
var  
  UP : UartPort;  
  YP : YmodemProtocol;  
...  
  UP.InitFast(Com1, 2400);  
...  
  if not YP.Init(@UP, True, False) then  
    {handle error} ;  
...  
  YP.Done;
```

Set up for a Ymodem file transfer at 2400 baud using Com1.

See Also

YmodemProtocol.InitCustom

InitYmodem

Syntax

```
constructor YmodemProtocol.InitCustom(APPtr : AbstractPortPtr;  
                                       Use1K, UseGMode : Boolean;  
                                       Options : Word);
```

Purpose

Allocate and initialize a protocol control block with options.

Description

This constructor instantiates an object of type YmodemProtocol in order to transfer files using the Ymodem protocol. The APPtr parameter specifies the port to be used. If Use1K is True, the protocol transmits data in 1024 byte blocks rather than 128 byte blocks. If UseGMode is True, the Ymodem G protocol is used rather than the regular Ymodem protocol. The Options parameter indicates which protocol options are in effect. Normally it is DefProtocolOptions (see APABSPCL/OPABSPCL in §7.C), in which case you can use the Init constructor instead of InitCustom.

Example

```
var  
  UP : UartPort;  
  YP : YmodemProtocol;  
...  
  UP.InitFast(Com1, 2400);  
...  
  if not YP.InitCustom(@UP, True, False, DefProtocolOptions) then  
    {handle error} ;  
...  
  YP.Done;
```

Set up for a Ymodem file transfer at 2400 baud using Com1.

See Also

YmodemProtocol.Init

InitCustomYmodem

InitCustomYmodem / InitYmodem

Syntax

```
procedure InitCustomYmodem(var P : ProtocolRecPtr; PortPtr : PortRecPtr;  
                           Use1K, UseGMode : Boolean; Options : Word);
```

Purpose

Allocate and initialize a Ymodem control block with the specified options.

Description

This routine is used to initialize a protocol record, to be used to transfer files under the Ymodem protocol. The P parameter specifies the port to be used. If Use1K is True, the protocol transmits data in 1024 byte blocks rather than 128 byte blocks. If UseGMode is True, the Ymodem G protocol is used rather than the regular Ymodem protocol. The Options parameter indicates which protocol options are in effect. Normally it is DefProtocolOptions (see APABSPCL/OPABSPCL in §7.C), in which case you can use InitYmodem instead of InitCustomYmodem.

Example

```
var  
  YP : ProtocolRecPtr;  
  P : PortRecPtr;  
...  
  InitPortFast(P, Com1, 2400);  
  ...  
  InitCustomYmodem(YP, P, True, False, DefProtocolOptions);  
  ...  
  DoneYmodem(YP);
```

Set up for a Ymodem file transfer at 2400 baud using Com1.

See Also

YmodemProtocol.InitCustom

InitYmodem

Syntax

```
procedure InitYmodem(var P : ProtocolRecPtr; PortPtr : PortRecPtr;  
                    Use1K, UseGMode : Boolean);
```

Purpose

Allocate and initialize a Ymodem control block.

Description

This routine initializes a protocol record for Ymodem file transfers using the default protocol options (DefProtocolOptions). See InitCustomYmodem for details.

Example

```
var  
  YP : ProtocolRecPtr;  
  P : PortRecPtr;  
...  
  InitPortFast(P, Com1, 2400);  
  ...  
  InitYmodem(YP, P, True, False);  
  ...  
  DoneYmodem(YP);
```

Set up for a Ymodem file transfer at 2400 baud using Com1.

Syntax

```
{IFDEF UseStreams}
constructor YmodemProtocol.Load(var S : IdStream);
```

Purpose

Load a YmodemProtocol object from a stream.

Description

This constructor loads a YmodemProtocol object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to YmodemProtocol.Store.

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Load is called.

See the entry for XmodemProtocol.Store for a discussion of how to link a protocol object loaded from a stream with a port object.

The stream registration routine for a YmodemProtocol object is YmodemProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Note that YmodemProtocol shares the same Store method with XmodemProtocol.

Example

```
uses
  OpRoot, ApPort, OoCom, OoAbsPcl, OoYmodem;
var
  UP : UartPort;
  YP : YmodemProtocol;
  S : BufIdStream;
  Status : Word;
begin
  if not S.Init('YMODEM.STM', SOpen, 1024) then begin
    WriteLn('Error opening stream'); Halt; end;
  S.RegisterHier(YmodemProtocolStream);
  S.RegisterHier(UartPortStream);
  S.RegisterPointer(1000, @UP);
  S.Get(UP);
  S.Get(YP);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Load error: ', Status); Halt; end;
  WriteLn('Loaded successfully');
  S.Done;
  YP.Done;
  UP.Done;
end.
```

Instantiates UP and YP from a stream created by the XmodemProtocol.Store example using the stream's Get method, which calls the objects' Load constructors indirectly.

Notice that a pointer to the port object is registered to ensure that YmodemProtocol.Load can create a link between the two objects.

F. Zmodem Protocol: APZMODEM/OOZMODEM

APZMODEM/OOZMODEM build on APABSCPL/OOABSPCL to provide the Zmodem protocol. Of all the protocols supported by Async Professional, Zmodem offers the best overall mix of speed, features, and tolerance for errors. The Zmodem protocol has many options and clearly was meant to have lots of room for growth. The Async Professional implementation of Zmodem does not cover the entire protocol specification. It implements those features most likely to be required by your application.

Zmodem was developed for the public domain by Chuck Forsberg under contract to Telenet. The purpose was to provide a durable protocol with strong error recovery features and good performance over a variety of network types (switched, satellite, etc.). It has generally achieved these design goals and, therefore, should be your protocol of choice whenever possible.

As with other protocols, Zmodem and all of its details and options will not be explained here. A simple outline (which is more than enough to use Zmodem in Async Professional) is provided here and you can see the ZMODEM.DOC file of PROTDOLZH for further details.

Zmodem borrows some concepts from Xmodem, Ymodem, and Kermit but is really a completely new protocol. Rather than the simple block structure of Xmodem and Ymodem, Zmodem employs headers, data subpackets, and frames. A header contains a header identifier, a type byte, four information bytes, and some block check bytes. A data subpacket contains up to 1024 data bytes, a data subpacket type identifier, and some block check bytes. A frame consists of one header and zero or more data subpackets.

Due to the complexity and variety of Zmodem header and data subpacket formats, they are not all detailed here. Instead, here is a high level look at a sample Zmodem file transfer:

Sender		Receiver	Explanation
'rz'<cr>	---		start indicator for automated transfers
ZrQinit	---		request for receiver's information
	<---	ZrInit	receiver answers with its options
ZFile	---		transmitter sends file information
	<---	ZrPos	receiver sets the starting file position
ZData	---		transmitter indicates file data to follow
data subpacket	---		transmitter sends a data subpacket
data subpacket	---		transmitter sends a data subpacket
...			...continues until all data sent
ZEof	---		transmitter indicates end-of-file
	<---	ZrInit	receiver indicates ready for next file
ZFin	---		transmitter indicates no-more-files
	<---	ZFin	receiver acknowledges
'00'	---		transmitter signs off

The ZXxx tags are the header types that the two computers trade back and forth as they decide what is to be done. You don't need to understand them unless you find yourself trying to decipher a Zmodem transfer using the Async Professional tracing facility. In such a case, you'll likely need to consult the ZMODEM.DOC file, the OOZMODEM source, or both. However, there shouldn't ever be a need to do either when using Async Professional.

In most cases all data in the file is sent in one ZData frame (the ZData header followed by as many data subpackets as required). The receiver doesn't *have* to acknowledge any of the blocks unless the transmitter specifically asks for an acknowledgement. Async Professional Zmodem never asks for an acknowledgement; however, it respects requests from the remote for acknowledgements.

Typically, once a file transfer is underway, the receiver interrupts the transmitter only if it receives a bad block (i.e., the received block check differs from the calculated block check). It does so by sending a ZrPos header, telling the transmitter where in the file to start retransmitting.

The protocol can be cancelled at any time by either side by sending five <CAN> characters (^X).

Control Character Escaping

Zmodem "escapes" certain control characters. In this context, escaping means that these characters are modified before being transmitted. This facilitates software flow control because it guarantees that flow control characters will never appear as part of the data. If such characters weren't escaped, flow control characters that happened to appear in the data stream would be misinterpreted by the modem or application and cause all sorts of problems.

Escaping isn't something you need to enable or disable because it's always on. It is mentioned here because escaping is what permits you to use software flow control with Zmodem. That isn't possible with the Xmodem/Ymodem family.

For your information, Zmodem escapes the following control characters:

cXon	- Xon character (\$11)
cXoff	- Xoff character (\$13)
cDle	- Zmodem escape character (\$10)
cXonHi	- Xon character with high bit set (\$91)
cXoffHi	- Xoff character with high bit set (\$93)
cDel	- Zmodem escape character with high bit set (\$90)

Zmodem Protocol Options

While the Zmodem specification describes all sorts of features and capabilities, not all Zmodem implementations are expected to support all of these features. One of the first things that happens in a Zmodem protocol is the receiver tells the transmitter what features it supports. The transmitter might then make some modifications to its standard behavior to accommodate the receiver's support (or lack of support) for a particular option. This process is automatic and doesn't require modification or adjustment by your program.

Since this process is handled automatically, you generally don't need to worry about it. For your information, however, the protocol options that Async Professional Zmodem provides and doesn't provide are listed here.

Async Professional Zmodem supports the following protocol options:

- can send and receive in true full duplex
- can receive data during disk I/O
- can send a break signal
- can use 32 bit CRCs

Async Professional honors a receiver's requests *except* for the following options:

- doesn't support encryption
- doesn't support LZ data compression
- doesn't escape all control characters
- doesn't escape the 8th bit

Conversion Options

The Zmodem specification describes a conversion option called "convert end-of-line." It requests that CR/LF pairs be replaced by Unix style newline characters or vice versa. Async Professional

does not currently support end-of-line conversions; it defaults to "no conversion." This means that Zmodem won't alter any incoming data.

The Zmodem specification has another conversion option called "recover/resume." This option is requested by the transmitter when it wants to resume a previously interrupted file transfer (assuming that the receiver kept the partial file from the interrupted transfer). When the receiver sees the request for this option, it compares the incoming file name with the files in its destination directory. If it finds the incoming file already exists and its copy of that file is smaller than the one being transmitted, it assumes that the transmitter only wants to transfer that portion of the file that the receiver doesn't already have.

When this condition exists, the receiver opens the existing file and moves the file pointer to the end of the file. It then tells the transmitter to move its file pointer to that same point in its copy of the file. The transmitter starts sending data from that point—meaning that the interrupted file transfer resumes from where it was interrupted.

You could also use this option when you need to update a remote copy of a file where the only change is that the source file has new data appended to it.

In either case, you specify this option as follows:

```
SetFileMask('BIGFILE');  
if ResumeFile then  
    SetRecoverOption(True);  
ProtocolTransmit;
```

Presumably, ResumeFile is a boolean that is True when the user has requested that an interrupted transfer should be resumed. If this is so, SetRecoverOption(True) is called to inform Zmodem of this fact. ProtocolTransmit tells the receiver that a recovery is in process and the receiver behaves as described above.

See FXO.PAS and FX.PAS for a working example of using the file recovery option.

File Management Options

Zmodem has a variety of file management options built into it. These are simple rules that tell Zmodem whether or not to accept a file. Here are the possible options:

```
WriteNewerLonger = 1; {Transfer if new, newer or longer}  
WriteCrc          = 2; {Not supported, same as WriteNewer}  
WriteAppend       = 3; {Transfer if new, append if exists}  
WriteClobber      = 4; {Transfer regardless}  
WriteNewer        = 5; {Transfer if new or newer}  
WriteDifferent    = 6; {Transfer if new or different dates/lengths}  
WriteProtect      = 7; {Transfer only if new}
```

Note that WriteCrc is *not* supported (this option requests that a file be transferred only if its CRC is different from the remote's copy of the file). See SetFileMgmtOptions for a more detailed description of these options and how to set them. The default is WriteNewer.

The file management options are always requested by the transmitter. Hence, when you are transmitting files via Zmodem, you should make a call to SetFileMgmtOptions with your desired options. If, for example, you want to transmit all files regardless of whether such files already exist on the remote, you should make the call:

```
SetFileMgmtOptions(False, False, WriteClobber);
```

Even though it's the transmitter that sets the management options, Async Professional allows the receiver to change these options. For example, suppose the transmitter has requested WriteClobber (which means it will overwrite existing files) even though you might want to accept only newer files. In this case the receiver would invoke the Override option of SetFileMgmtOptions:

```
SetFileMgmtOptions(True, False, WriteNewer);
```

This tells Async Professional Zmodem to ignore the file management options requested by the transmitter (by passing True for Override, the first parameter) and use WriteNewer instead.

Another file management option is called SkipNoFile; it is the second boolean parameter to SetFileMgmtOptions. Use this option to force the receiver to skip a file if the receiver doesn't already have a file of the same name.

The receiver applies the specified file management rules and either accepts the file (in which case the file transfer proceeds normally) or rejects it. The receiver rejects the file by sending a ZSkip frame to the transmitter. The transmitter understands this to mean the receiver doesn't want the current file; it skips sending the file and moves on to the next file to transmit.

Finally, don't forget that you can implement your own file management rules with a custom AcceptFile function. See "AcceptFile Function" in §7.C for more details.

Automatic Block Size Control

APZMODEM and OOZMODEM ProtocolTransmit procedures decrease and increase the number of bytes transmitted per block in response to retransmission requests from the receiver (usually due to poor line conditions or random line errors). The rationale behind this is that more small blocks will get through without errors since there's less time for the block to be hit by line noise. And, even if the small block is corrupted by line errors, it is faster to retransmit a small block than a large block.

ProtocolTransmit employs the following logic in controlling the block size: If it receives an unsolicited request from the receiver to retransmit old data, it reduces the block size from 1024 to 512 and starts retransmitting data from the requested position. If ProtocolTransmit receives another retransmit request, it reduces the block size from 512 to 256. It never reduces the block size below 256 bytes. The transmitter raises the block size back to 1024 when it sends four blocks in a row without receiving any requests to retransmit old blocks.

Note that similar behavior is also used with 8K Zmodem. The block size is divided by two (down to the limit of 256 bytes) for each retransmission request received. The block size is raised all the way back to 8K after four blocks are transmitted without any retransmission requests.

Block size control is automatic and cannot be disabled. While this behavior is not documented in the public domain Zmodem specification, it is the process followed by the very popular DSZ program and is acceptable to any standard Zmodem implementation.

Large Block Support

APZMODEM and OOZMODEM also include support for large (8K) blocks. This support is outside the public domain specification and was added largely for Async Professional users who need to transfer files with several popular BBS and FIDONet mailer programs. Since large blocks are not supported by other "standard" Zmodem implementations, use of large blocks is not automatic—you must specifically enable support for them when you initialize the protocol. In addition, you must turn this feature on and off using SetBigSubpacketOption before each call to ProtocolTransmit or ProtocolReceive.

The concerns mentioned in the APABSPCL/OOABSPCL section regarding port output buffer size become even more critical when using the large buffers required for 8K support. The output buffer size must be large enough to hold an entire outgoing Zmodem block. Because Zmodem transforms certain binary control characters into an escape character and a modified control character, a block could double in size during this escaping process if every byte required escaping.

When using APUART, you must make your port output buffer large enough. In the case of 8K Zmodem, your output buffers must be 16K bytes plus 30 for overhead ($16384+30 = 16414$). If your output buffer is too small, the protocol initialization routines (Init, InitZmodem, etc.) will fail with the error code ecBuffersTooSmall.

When you want to use 8K Zmodem blocks, you need to prepare early by specifying the option apZmodem8K when instantiating the protocol object or initializing the protocol record. This tells the protocol to allocate buffers large enough to handle the 8K blocks and verifies that your output buffer is at least 16414 bytes in size. Specifying this option tells the protocol to allocate the large buffers, but does not turn on support for 8K blocks. Use the SetBigSubpacketOption procedure to activate 8K Zmodem before calling ProtocolTransmit or ProtocolReceive.

The Zmodem 8K option can be specified only when the protocol is initialized. Attempts to set it or turn it off via apOptionsOn or apOptionsOff are ignored.

Shared Declarations

Constants

DefFinishWait : Word = 364;

Default length of time (in clock ticks) to wait after a ZFin packet for a response from the remote. The procedural version of this constant has the name DefFinishWaitZM.

DefFinishRetry : Word = 3;

Default number of times to resend the ZFin if no response is received from the remote. This is used only when receiving. The procedural version of this constant has the name DefFinishRetryZM.

DefReceiveTimeout : Word = 364;

Default ticks for received data (20 seconds). This describes how long Zmodem will routinely wait for the next character in a sequence. Once Zmodem starts receiving a header or a data subpacket, it will wait no more than DefReceiveTimeout ticks for each successive character before assuming an error has occurred (such as a broken connection).

DefReceiveTimeout also describes how long Zmodem will wait for a new header before taking a recovery action—such as resending the last header.

This constant does not control how long Zmodem will wait for the initial handshaking packet. That is controlled by SetHandshakeWait (in APABSPCL/OOABSPCL).

DrainingStatusInterval : Word = 18;

Interval between calls to the ShowStatus hook. This is used only while the output buffer is draining at the end of a file transmit (18 ticks = 1 second).

HandshakeWait = 1092;

Default time to wait (in clock ticks) for a response during handshaking (1092 clock ticks = 60 seconds). See the entry for SetHandshakeWait (APABSPCL/OOABSPCL). The procedural version of this constant has the name HandshakeWaitZM.

MaxBadBlocks = 20;

If this many consecutive errors are received, the protocol is aborted.

WriteNewerLonger = 1; {Transfer if new, newer or longer}
WriteCrc = 2; {Not supported, same as WriteNewer}
WriteAppend = 3; {Transfer if new, append if exists}

```

WriteClobber      = 4; {Transfer regardless}
WriteNewer        = 5; {Transfer if new or newer}
WriteDifferent    = 6; {Transfer if new or different dates/lengths}
WriteProtect      = 7; {Transfer only if new}

```

These constants are used to specify the desired file management options governing received files. The default setting is WriteNewer. See SetFileMgmtOptions later in this section and "File Management Options" earlier in this section for details.

```
ZmodemOverHead : Word = 20;
```

This is an estimate of the average number of overhead bytes per data subpacket (required by EstimateTransferSecs to estimate the time required to transfer a specified number of bytes). Because Zmodem escapes various control characters, the actual overhead will vary between data subpackets. Twenty is a reasonably accurate estimate. However, you can change this value if you know that your data will have different overhead.

```
ZmodemTurnDelay : Word = 0;
```

This is an estimate of the turnaround delay in a Zmodem protocol (required by EstimateTransferSecs to estimate the time required to transfer a specified number of bytes). Turnaround delay is the time, in milliseconds, required for the remote system to acknowledge each block. Generally, Zmodem doesn't have any turnaround delay at all, since its data subpackets do not require an acknowledgement. However, some Zmodem implementations may request such acknowledgements. You may need to increase ZmodemTurnDelay when connecting to such systems.

ZrQinit	= #0;	{Request init (to receiver)}
ZrInit	= #1;	{Init (to sender)}
ZsInit	= #2;	{Init (to receiver) (optional)}
ZAck	= #3;	{Acknowledge last frame}
ZFile	= #4;	{File info frame (to receiver)}
ZSkip	= #5;	{Skip to next file (to transmitter)}
ZNak	= #6;	{Error receiving last data subpacket}
ZAbort	= #7;	{Abort protocol}
ZFin	= #8;	{Finished protocol}
ZRpos	= #9;	{Resume from this file position}
ZData	= #10;	{Data subpacket(s) follows}
ZEOF	= #11;	{End of current file}
ZFerr	= #12;	{Error reading or writing file}
ZCRC	= #13;	{Request for file CRC (to receiver)}
ZChallenge	= #14;	{Challenge the sender}
ZCompl	= #15;	{Complete}
ZCan	= #16;	{Cancel requested (to either)}
ZFreeCnt	= #17;	{Request diskfree}
ZCommand	= #18;	{Execute this command (to receiver)}

For information only. These are the various frame types used by Zmodem.

APZMODEM Declarations

Types

```

ZmodemPtr = ^ZmodemProtocol;
ZmodemProtocol =
    record
        ...
    end;

```

Record used to store data needed by the various routines in APZMODEM to implement the Zmodem protocol.

Procedures and Functions

```
procedure PrepareReceivePartZM(P : ProtocolRecPtr);
  {-Prepare to call the background function ProtocolReceivePartZM}
procedure PrepareTransmitPartZM(P : ProtocolRecPtr);
  {-Prepare to use the background function ProtocolTransmitPartZM}
procedure ProtocolReceiveZM(P : ProtocolRecPtr);
  {-Receive a batch of files transmitted using Zmodem}
function ProtocolReceivePartZM(P : ProtocolRecPtr) : ProtocolStateType;
  {-Perform one increment of a background protocol receive}
procedure ProtocolTransmitZM(P : ProtocolRecPtr);
  {-Start Zmodem protocol transmit}
function ProtocolTransmitPartZM(P : ProtocolRecPtr) : ProtocolStateType;
  {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the similarly named functions in APABSPCL (for example, PrepareReceivePartZM is documented in PrepareReceivePart). See §7.C for a complete discussion of these functions.

OOZMODEM Declarations

Types

```
ZmodemProtocolPtr = ^ZmodemProtocol;
ZmodemProtocol =
  object(AbstractProtocol)
  ...
end;
```

Object that defines data fields and methods needed to implement the Zmodem protocol.

Procedures and Functions

```
procedure ZmodemProtocol.PrepareReceivePart; virtual;
  {-Prepare to call the background function ProtocolReceivePart}
procedure ZmodemProtocol.PrepareTransmitPart; virtual;
  {-Prepare to use the background function ProtocolTransmitPart}
procedure ZmodemProtocol.ProtocolReceive; virtual;
  {-Receive a batch of files transmitted using Zmodem}
function ZmodemProtocol.ProtocolReceivePart : ProtocolStateType; virtual;
  {-Perform one increment of a background protocol receive}
procedure ZmodemProtocol.ProtocolTransmit; virtual;
  {-Start Zmodem protocol transmit}
function ZmodemProtocol.ProtocolTransmitPart : ProtocolStateType; virtual;
  {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the ones with the same name in OOABSPCL. See §7.C for a complete discussion.

Syntax

destructor ZmodemProtocol.Done; virtual;

Purpose

Dispose of the protocol record.

Description

This destructor disposes of the data structures allocated on the heap by the Init constructor and destroys the object.

Example

See the example for InitCustom.

Syntax

procedure DoneZmodem(var P : ProtocolRecPtr);

Purpose

Dispose of the protocol record.

Description

This routine disposes of the protocol record pointed to by P, previously allocated by a call to InitZmodem or InitCustomZmodem, as well as the data structures that were allocated on the heap.

Example

See the examples for InitCustomZmodem and InitZmodem.

See Also

InitZmodem

Init / InitCustom

Syntax

constructor ZmodemProtocol.Init(APPtr : AbstractPortPtr);

Purpose

Allocate and initialize a protocol control block.

Description

This constructor initializes a ZmodemProtocol object using the default protocol options (DefProtocolOptions). See InitCustom for details.

Example

```
var
  UP : UartPort;
  ZP : ZmodemProtocol;
...
  UP.InitFast(Com1, 2400);
...
  if not ZP.Init(@UP) then
    {handle error} ;
...
  ZP.Done;
```

Set up for a Zmodem file transfer at 2400 baud using Com1.

See Also

ZmodemProtocol.InitCustom

InitZmodem

Syntax

constructor ZmodemProtocol.InitCustom(APPtr : AbstractPortPtr; Options : Word);

Purpose

Allocate and initialize a protocol control block with the specified options.

Description

This constructor initializes a ZmodemProtocol object for file transfers using the specified port and protocol Options (normally DefProtocolOptions). It also allocates space on the heap for some large data structures needed for internal use (approximately 11KB).

Example

```
var
  UP : UartPort;
  ZP : ZmodemProtocol;
...
  UP.InitFast(Com1, 2400);
...
  if not ZP.InitCustom(@UP, DefProtocolOptions) then
    {handle error} ;
...
  ZP.Done;
```

Set up for a Zmodem file transfer at 2400 baud using Com1.

See Also

InitCustomZmodem

Syntax

```
procedure InitCustomZmodem(var P : ProtocolRecPtr; PortPtr : PortRecPtr;  
                           Options : Word);
```

Purpose

Allocate and initialize a protocol control block with the specified options.

Description

This routine initializes a protocol record for Zmodem file transfers using the specified port and protocol Options (normally DefProtocolOptions). It also allocates space on the heap for some large data structures needed for internal use (approximately 11KB).

Example

```
var  
  ZP : ProtocolRecPtr;  
  P : PortRecPtr;  
...  
  InitPortFast(P, Com1, 2400);  
  ...  
  InitCustomZmodem(ZP, P, DefProtocolOptions);  
  ...  
  DoneZmodem(ZP);
```

Set up for a Zmodem file transfer at 2400 baud using Com1.

See Also

ZmodemProtocol.InitCustom

Syntax

```
procedure InitZmodem(var P : ProtocolRecPtr; PortPtr : PortRecPtr);
```

Purpose

Allocate and initialize a protocol control block.

Description

This routine initializes a protocol record for Zmodem file transfers using the default protocol options (DefProtocolOptions). See InitCustomZmodem for details.

Example

```
var  
  ZP : ProtocolRecPtr;  
  P : PortRecPtr;  
...  
  InitPortFast(P, Com1, 2400);  
  ...  
  InitZmodem(ZP, P);  
  ...  
  DoneZmodem(ZP);
```

Set up for a Zmodem file transfer at 2400 baud using Com1.

See Also

ZmodemProtocol.Init

InitCustomZmodem

Load

Syntax

```
{IFDEF UseStreams}  
  constructor ZmodemProtocol.Load(var S : IdStream);
```

Purpose

Load a ZmodemProtocol object from a stream.

Description

This constructor loads a ZmodemProtocol object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to ZmodemProtocol.Store.

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Load is called.

See the entry for XmodemProtocol.Store for a discussion of how to link a protocol object loaded from a stream with a port object.

The stream registration routine for a ZmodemProtocol object is ZmodemProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses  
  OpRoot, ApPort, OoCom, OoAbsPcl, OoZmodem;  
var  
  UP : UartPort;  
  ZP : ZmodemProtocol;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not S.Init('ZMODEM.STM', SOpen, 1024) then begin  
    WriteLn('Error opening stream');  
    Halt;  
  end;  
  S.RegisterHier(ZmodemProtocolStream);  
  S.RegisterHier(UartPortStream);  
  S.RegisterPointer(1000, @UP);  
  S.Get(UP);  
  S.Get(ZP);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Load error: ', Status);  
    Halt;  
  end;  
  WriteLn('Loaded successfully');  
  S.Done;  
  ZP.Done;  
  UP.Done;  
end.
```

Instantiates UP and ZP from a stream created by the Store example using the stream's Get method, which calls the objects' Load constructors indirectly.

Notice that a pointer to the port object is registered to ensure that ZmodemProtocol.Load can create a link between the two objects.

Syntax

```
procedure SetBigSubpacketOption(P : ProtocolRecPtr; UseBig : Boolean);  
procedure ZmodemProtocol.SetBigSubpacketOption(UseBig : Boolean);
```

Purpose

Activate 8K Zmodem.

Description

You can use this procedure to turn on 8K Zmodem if you specified apZmodem8K when you instantiated the protocol object or initialized the protocol record.

Specify True for UseBig to activate the use of 8K blocks. Thereafter, ProtocolTransmit will transmit using 8K blocks and ProtocolReceive will be able to receive blocks up to 8K in length.

Specify False for UseBig to deactivate 8K blocks. Thereafter, ProtocolTransmit will use only the standard 1024 byte block and ProtocolReceive will be able to receive blocks only up to 1024 bytes in length.

Note that if you didn't specify ap8KZmodem when the protocol was instantiated or initialized, all calls to SetBigSubPacketOption are ignored.

SetFileMgmtOptions

Syntax

```
procedure SetFileMgmtOptions(P : ProtocolRecPtr;  
                             Override, SkipNoFile : Boolean; FOpt : Byte);  
procedure ZmodemProtocol.SetFileMgmtOptions(Override, SkipNoFile : Boolean;  
                                             FOpt : Byte);
```

Purpose

Set file management options.

Description

This routine allows you to tell the protocol what Zmodem file management rules you want it to apply when receiving a file that already exists.

Override is used only when you are receiving files. When True, it means that the transmitter's requested file management options will be ignored and the receiver will use the option specified in FOpt.

SkipNoFile also applies only when receiving files. When True, it means that the receiver should skip all incoming files that don't already exist.

The FOpt parameter must be one of the following:

WriteNewerLonger	Overwrite the existing file if the incoming file is new, or newer or longer than the existing file
WriteCrc	An unsupported Zmodem option, treated the same as WriteNewer
WriteAppend	Write the file if it is new; if not, append the transmitted file to the end of the existing one
WriteClobber	Write the transmitted file no matter what
WriteNewer	Write the file if it is new or newer than the existing file
WriteDifferent	Write the file if it is new, or if its date or size is different from the existing file's
WriteProtect	Write the file only if it is new

If SetFileMgmtOptions is not called, the defaults are WriteNewer, Override = False, and SkipNoFile = False.

See "File Management Options" earlier in this section for more information on how and why you might use these options.

Examples

```
var  
  ZP : ProtocolRecPtr;  
...  
  SetFileMgmtOptions(ZP, False, False, WriteClobber);  
  SetFileMask(ZP, '*.PAS');  
  ProtocolTransmitZM(ZP);
```

All .PAS files in the current directory are transmitted to the remote computer. They overwrite any existing .PAS files of the same name.

```
ProtocolReceiveZM(ZP);
```

Receives all files sent by the transmitter using the file management rules specified by the transmitter.

```
SetFileMgmtOptions(ZP, True, False, WriteProtect);  
ProtocolReceiveZM(ZP);
```

Ignores the transmitter's file management rules and uses WriteProtect instead (which means that existing files are not overwritten).

See Also

SetOverwriteOption (APABSPCL)
AbstractProtocol.SetOverwriteOption

SetAcceptFileFunc (APABSPCL)
AbstractProtocol.SetAcceptFileFunc

Syntax

```
procedure SetFinishWaitZM(P : ProtocolRecPtr; NewWait : Word; NewRetry : Byte);  
procedure ZmodemProtocol.SetFinishWait(NewWait : Word; NewRetry : Byte);
```

Purpose

Set new finish wait and retry values.

Description

The Zmodem specification indicates that the appropriate action after a ZFin response timeout is to exit without an error since the protocol is over anyway. However, this strategy does not work well with DSZ, a ZModem implementation by Omen Technology, Inc. DSZ does retries after a ZFin response timeout, which can sometimes cause a few extra packet transfers when the protocol is over, if you have a low handshake value (about 10 seconds or less). To handle this situation, Async Professional mimics DSZ instead of following the Zmodem specification. When receiving files, Async Professional retries the ZFin up to DefFinishRetry times and exits with no error if no response is received. When transmitting, Async Professional waits DefFinishWait seconds for a response.

Use this procedure to change the ZFin response time wait and ZFin retry values. NewWait is the new wait time, in ticks, to wait for a response from the remote to a ZFin. Specify zero for this parameter if you don't want to change it. NewRetry is the number of times to resend the ZFin if a response is not received. Unlike NewWait, passing a zero for NewRetry does *not* prevent a change, but instead sets the number of retries to zero.

Example

```
SetFinishWaitZM(ZP, Secs2Tics(15), 3);
```

Changes the wait time to 15 seconds and sets the number of retries to 3.

Syntax

```
procedure SetRecoverOption(P : ProtocolRecPtr; OnOff : Boolean);  
procedure ZmodemProtocol.SetRecoverOption(OnOff : Boolean);
```

Purpose

Enable or disable file recovery.

Description

Zmodem is capable of resuming interrupted file transfers if the receiver kept the partial file when the previous transfer was interrupted. The transmitter requests this action by calling SetRecoverOption(True). The request is transmitted to the receiver along with the file name to be recovered. If the receiver has such a file already, it sends back the current size of the file and the transmitter adjusts the file pointer in its copy of the file and starts sending data from that point.

If the receiver doesn't already have such a file, then a normal file transfer will take place.

Although this option is typically requested by the transmitter, Async Professional permits the receiver to request this option as well.

See "Conversion Options" earlier in this section for more information.

Example

```
SetFileMask(ZP, 'BIGFILE');  
if ResumeFile then  
  SetRecoverOption(ZP, True);  
ProtocolTransmitZM(ZP);
```

Assume that a previous attempt to transmit BIGFILE failed before the entire file was transmitted and ResumeFile is a boolean set by the program when the user asks to attempt a recovery. If the partial file still exists on the remote computer, then the file transfer will resume at the interrupted point. When using objects, the ZP parameters would be omitted and ProtocolTransmit would be used instead of ProtocolTransmitZM.

Store

Syntax

```
($IFDEF UseStreams)
procedure ZmodemProtocol.Store(var S : IdStream);
```

Purpose

Store a ZmodemProtocol object to a stream.

Description

Store writes an image of the ZmodemProtocol object to the stream S.

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Store is called.

More importantly, you must give ZmodemProtocol.Store some means of forming a link between the protocol object and the port object whose address was passed to the Init constructor. See XmodemProtocol.Store for a discussion of this issue.

The stream registration routine for a ZmodemProtocol object is ZmodemProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses
  OpRoot, ApPort, OoCom, OoAbsPcl, OoZmodem;
var
  UP : UartPort;
  ZP : ZmodemProtocol;
  S : BufIdStream;
  Status : Word;
begin
  if not UP.InitFast(Com1, 2400) then begin
    WriteLn('Error initializing port');
    Halt;
  end;
  if not ZP.Init(@UP) then begin
    WriteLn('Error initializing protocol');
    Halt;
  end;
  if not S.Init('ZMODEM.STM', SCreate, 1024) then begin
    WriteLn('Error initializing stream');
    Halt;
  end;
  S.RegisterHier(ZmodemProtocolStream);
  S.RegisterHier(UartPortStream);
  S.RegisterPointer(1000, @UP);
  S.Put(UP);
  S.Put(ZP);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error: ', Status);
    Halt;
  end;
  WriteLn('Stored successfully');
  S.Done;
  ZP.Done;
  UP.Done;
end.
```

Instantiates the port and protocol objects and then stores them in a stream using the stream's Put method, which calls the objects' Store methods indirectly.

A pointer to the UartPort object is registered so that ZmodemProtocol.Store can create a link between the two objects.

See ZmodemProtocol.Load for an example program that reloads these objects.

See Also

ZmodemProtocol.Load

XmodemProtocol.Store

G. Kermit Protocol: APKERMIT/OOKERMIT

The Kermit protocol was developed to facilitate file transfers in environments that other protocols can't handle. Such environments include links that only pass 7 data bits, links that can't handle control characters, computer systems that can't handle large blocks, and other diverse links such as those between a PC and a mainframe.

Kermit is a public domain protocol that was developed at Columbia University in New York City. (The name Kermit refers to Kermit the Frog, from the television program, The Muppet Show.) A complete description of the Kermit protocol is beyond the bounds of this manual. However, the simple description provided here is more than sufficient for using the Async Professional implementation of the Kermit protocol. If you are interested in getting the official description of the Kermit protocol, write to Columbia University, Kermit Distribution, Department OP, 612 West 115th Street, NY, NY, 10025.

Character Quoting

Character quoting means pretty much the same thing that escaping means in Zmodem. "Quoting" a character means that the character is replaced by a quote character and a modified form of the original character. The quote character tells the receiver how to convert the modified character back into the original character.

The purpose of quoting is to make sure that certain binary characters are never put into the data stream. Those binary characters might be wrongly interpreted by a modem or another part of the serial link.

Although Zmodem transforms only a few critical characters (notably, the Xon/Xoff flow control characters), Kermit quotes nearly all characters. This is one of the features that permits Kermit to run in nearly any environment. When all the quoting is finished, a Kermit packet (Kermit calls its chunks of data "packets" rather than blocks or frames) consists almost entirely of printable ASCII characters. The only exceptions are the <SOH> character at the start of each packet and the <CR> at the end.

Kermit accomplishes this by quoting all control characters. Each control character becomes a quote character and a modified control character. For example, ^A becomes '#A' ('#' is the quote character). Kermit calls this process "Ctl" and it works like this:

```
Ctl(x) = x xor $40;
```

This operation is its own inverse, that is, Ctl(Ctl(x)) = x.

Kermit also quotes characters with their 8th bit set, which is how it manages to transmit 8 bit data over 7 bit data links. The quote character for the 8th bit is '&'. So, for example, the 'A' character with its high bit set (\$C1) becomes '&A'.

Kermit has a very simple built-in data compression mechanism called run length encoding. If it sees a long string of repeated characters, it compresses the string into a quote character, a length byte, and the repeated character. Obviously, this means that there must be at least 4 repeated characters before there is any gain by run length encoding. You can control this threshold in Async Professional with the DefMinRepeatCnt typed constant. The quote character for run length encoding is '~'. So, the string 'AAAAA' becomes '~%A' (the length is transformed by the "ToChar" operation, described next).

Finally, some of the numbers in the Kermit header (and in repeated character strings) need to be transformed into printable characters. This is achieved by adding 32 to the number before it is

inserted into the header and subtracting 32 when the receiver extracts the number from the header. In Kermit parlance, these operations are referred to as "ToChar" and "UnChar."

Kermit Packets

The following diagram shows the general format of a Kermit packet:

<SOH>	<length>	<seq>	<type>	<up to 91 bytes of data>	<check>	<term>
-------	----------	-------	--------	--------------------------	---------	--------

The <SOH> character, also called the mark field, indicates the start of a Kermit packet.

The length byte specifies the number of bytes that follow. Since this must be a printable character, its range is limited to 94 different values, meaning the maximum length of a normal Kermit packet is 96 bytes (including the <SOH> and the <length> field).

The <seq> byte is the sequence number. This is in the range of 0 to 63—after 63 it cycles back to 0.

The <type> byte describes the various Kermit packet types (analogous to the Zmodem frame types).

The data field is up to 91 bytes *including* all quote characters. The number of actual data bytes could be considerably less, particularly if binary data is being transmitted.

The standard Kermit <check> field is a single-byte checksum. Kermit offers two optional block check methods called two-byte checksum and three-byte-CRC. Both are available in Async Professional (see SetKermitCheck).

The <term> character is the packet terminator and is carriage return (ASCII 15) by default. You will probably never need to change the terminator.

A typical Kermit protocol transfer might look like this:

Transmitter		Receiver	Explanation
KSendInit	---->		Transmitter sends its options
	<---	KAck	Receiver answers with its options
KFile	---->		Transmitter sends filename
	<---	KAck	Receiver acknowledges filename
KData	---->		Transmitter sends data packet
	<---	KAck	Receiver acknowledges data packet
KData	---->		Transmitter sends data packet
	<---	KAck	Receiver acknowledges data packet
...			...continues until all data sent
KEndoffile	---->		Transmitter says end of file
	<---	KAck	Receiver acknowledges and closes file
KBreak	---->		Transmitter says end of protocol
	<---	KAck	Receiver acknowledges end of protocol

The KXxx tags are the packet types that the two computers exchange as they decide what is to be done. You don't really need to understand them unless you find yourself trying to decipher a Kermit transfer using the Async Professional tracing facility. In such a case, you'll likely need to consult the official Kermit Protocol Manual, the APKERMIT/OOKERMIT source, or both. There should never be a need to do this when using Async Professional.

Kermit Options

Like Zmodem, Kermit offers a variety of options. An implementation of Kermit is not required to support all options. Therefore one of the first things that happens in a Kermit protocol is that the

two sides exchange their desired options and use the "lowest common denominator" between the two.

Here are the options Async Professional supports and the default values it uses.

MaxPacketLen	(80 bytes)	maximum length of the data field
MaxTimeout	(5 seconds)	maximum timeout between characters
PadCount	(0 bytes)	no pad characters before packets
PadChar	(#0)	no pad character is used
Terminator	(<CR>)	packet terminator is a carriage return
CtlPrefix	(' #')	control character prefix is '#'
HibitPrefix	('Y')	honor 8bit quoting but don't require it
Check	('1')	use a 1 byte checksum
RepeatPrefix	('~')	repeat prefix is '~'
CapabilitiesMask	(0)	no extended capabilities
WindowSize	(0)	no windows
MaxLongPacketLen	(0)	no long packets

MaxPacketLen is the maximum number of bytes you want Kermit to include in one packet. The maximum value is 91; the default value is 80. You would generally want this number to be as high as possible (though the Kermit Protocol Manual suggests 80 as the default).

MaxTimeout is the amount of time, in seconds, that you want Kermit to wait for a new packet or the next byte of data in a sequence. If more than MaxTimeout seconds elapse without a response, Kermit assumes an error occurred and it resends its last packet.

PadCount and PadChar describe the padding that can be added to the front of all Kermit packets. The only reason you might need padding is if the remote machine needs a delay between sending you a packet and being ready to accept a packet itself. In this case, you would specify enough padding characters to accommodate that machine's delay. Generally, though, you won't need to use padding at all. Async Professional Kermit automatically honors a remote's request for padding.

Terminator is the last character, following the check character(s), in a packet. Although all Kermit packets have a terminator, it's used only by systems that need an end-of-line character before they can start processing input.

CtlPrefix is the control character prefix that Kermit uses when performing "Ctl" quoting (to transform control characters into printable ASCII characters). Generally, you won't need to change this prefix.

HibitPrefix is the high-bit character prefix that Kermit uses when transforming high-bit characters (data bytes with bit 7 set) into characters without the high-bit set. Generally, you won't need to change this prefix. See SetHibitPrefix for more information on the high-bit character prefix.

Check specifies the type of block checking Kermit is to perform. '1' means that Kermit should use a single-byte checksum. All Kermit implementations are guaranteed to support this form of block checking. A '2' means that Kermit should use a two-byte checksum (which offers only slightly more protection than the single-byte checksum). '3' means that Kermit should use a three-byte CRC. This is the preferred block check method because it offers the highest level of error detection. Unfortunately, not all Kermit implementations support block check methods '2' and '3'. If the remote computer doesn't support the block check method you request, both sides drop back to '1' (the single-byte checksum).

RepeatPrefix is the repeated-character prefix that Kermit uses when compressing long strings of repeated characters. Generally, you won't need to change this prefix.

CapabilitiesMask is a bit mask field that describes which extended capabilities, if any, are requested. Async Professional supports two extended capabilities: sliding windows and long blocks. CapabilitiesMask is automatically filled in depending on which capabilities you select.

WindowSize is the number of windows requested when sliding windows is enabled with a call to SetMaxWindows.

MaxLongPacketLen is the maximum long packet length requested when long block support is enabled with a call to SetMaxLongPacketLen.

The two sides of a Kermit protocol automatically negotiate which options to use, so no intervention is required by your program. However, if you wish to change the default options, use InitCustom or SetKermitOptions, supplying your own KermitOptionRec with appropriate values.

Async Professional Kermit does not provide Kermit server functions and does not support file attribute packets.

Long Packets

APKERMIT and OOKERMIT include support for "long packets." A long packet is an extension to standard Kermit that permits data packets of up to 1024 bytes. Long packets can substantially improve protocol throughput on relatively clean connections that have small turnaround delays. To keep within the design intentions for long packets as expressed in the Kermit Protocol Manual, long packet support is turned off by default and must be enabled via SetMaxLongPacketLen. In most cases it will be disabled by default at the remote end also.

While the specification allows for packets up to 9024 bytes, Async Professional limits long packets to 1024 bytes. Packets longer than 1024 bytes do not appreciably increase throughput, but they dramatically increase retransmission time when a line error occurs.

The specification also recommends the use of the higher-order checksums with long packets, but it is not required. Async Professional defaults to 2 byte checksums when long packet support is enabled, but drops back to 1 byte checksums if requested to do so by the remote Kermit.

Sliding Windows Control

Async Professional Kermit supports the extension known as Sliding Windows Control, also called "SuperKermit." Sliding Windows Control (SWC) provides a send-ahead facility that can dramatically improve protocol throughput under conditions where turnaround delays tend to be large, such as occurs when using satellite links.

"Send-ahead" means that the transmitter sends many blocks without waiting for an acknowledgement for each block. It collects acknowledgements when they eventually arrive and marks the previously transmitted blocks as acknowledged. This reduces turnaround delay (the time it takes the receiver to send an acknowledgement) to zero—significantly improving throughput.

SWC works by keeping a circular table of transmitted packets—the maximum number of packets in this table is called the "window size." The window size is selected at runtime and must be between 0 and 31 (0 meaning no sliding window support). If the transmitter and receiver specify different window sizes, the smaller of the two is used.

As each packet is transmitted, it is added to the table. When an acknowledgment is eventually received for that packet, the table is rotated (i.e., the oldest acknowledged packet is discarded). If the table fills, the transmitter does not send any more packets until it receives acknowledgements for one or more existing packets.

When receiving data, each packet is kept in the table until the table is full. Then the oldest packet is written to disk. If packets are garbled or missing, the receiver sends <NAK>s for all of the bad packets at once.

It is possible to mix long packets and SWC, but memory consumption rises dramatically. In the worst case you could have 1024 byte packets and 31 windows, which requires a 31744 byte buffer to hold incoming/outgoing packets (as opposed to the default 80 byte buffer). Sliding window support is off by default.

Shared Declarations

Constants

```
DefHibitPrefix : Char = '&';
```

Default character for high bit prefixing.

```
DefKermitOptions : KermitOptionRec = (
  MaxPacketLen : 80;      {80 characters}
  MaxTimeout : 5;        {5 seconds}
  PadCount : 0;           {No pad chars}
  PadChar : #0;           {Null pad char}
  Terminator : cCR;       {Carriage return}
  CtlPrefix : '#';        {'#' char}
  HibitPrefix : 'Y';      {'Y' means don't need high bit prefixing}
  Check : 'I';            {I byte checksum}
  RepeatPrefix : '~';     {Default repeat prefix}
  CapabilitiesMask : 0;    {No default extended capabilities}
  WindowSize : 0;         {No default windows}
  MaxLongPacketLen : 0;   {No default long packets}
```

Default Kermit options, based on the Kermit Protocol Manual.

```
DefMinRepeatCnt : Byte = 4;
```

Sets the minimum repeated character threshold for Kermit's built-in run length encoding of repeated character strings. For the default value of four, Kermit will not apply run length encoding unless it sees at least four repeated characters. Generally, you won't need to adjust this value.

```
FastAbort : Boolean = False;
```

The Kermit protocol manual describes the accepted process for aborting a protocol. Some implementations of Kermit (notably, ProComm) use a method that is faster but might not be recognized by all Kermit implementations (it just sends an error packet and exits). Setting FastAbort to True enables this "ProComm" style of aborting.

KBreak	= 'B';	{Break transmission (EOT)}
KData	= 'D';	{Data packet}
KError	= 'E';	{Error packet}
KFile	= 'F';	{File header packet}
KNak	= 'N';	{Negative acknowledge packet}
KSendInit	= 'S';	{Initial packet (exchange options)}
KDisplay	= 'X';	{Display text on screen packet}
KAck	= 'Y';	{Acknowledge packet}
KEndOfFile	= 'Z';	{End of file packet}

For information only. These are the Kermit packet types that Async Professional understands.

```
KermitOverhead : Word = 20;
```

This is an estimate of the average number of overhead bytes per packet (required by EstimateTransferSecs to estimate the time required to transfer a specified number of bytes). Because Kermit quotes all control characters, the actual overhead can vary between data packets. You might need to increase this number if you are transferring binary files.

KermiTurnDelay : Word = 1000;

This is an estimate of the turnaround delay in a Kermit protocol (required by EstimateTransferSecs to estimate the time required to transfer a specified number of bytes). Turnaround delay is the time, in milliseconds, required for the remote system to acknowledge the last block. 1000 milliseconds is a reasonable general-purpose estimate. If you are using Sliding Windows Control, SWCKermiTTurnDelay (see below) is used instead of KermiTTurnDelay.

MaxWindowSlots = 27;

Maximum number of table slots (window size) for SWC. By default MaxWindowSlots is set to 27 instead of the specification's stated limit of 31. This avoids a bug in the MSKERMIT program when it tries to use window sizes above 27.

SWCKermiTTurnDelay : Word = 5;

Turnaround delay on Kermit Sliding Windows Control transfers.

Types

```
KermiTOptionRec =
  record
    MaxPacketLen      : Byte; {maximum length of the data field}
    MaxTimeout        : Byte; {maximum timeout between characters (secs)}
    PadCount          : Byte; {number of pad characters before packets}
    PadChar           : Char; {pad character}
    Terminator        : Char; {packet terminator}
    CtlPrefix         : Char; {control character prefix}
    HbitPrefix        : Char; {8bit quoting prefix}
    Check             : Char; {check method}
    RepeatPrefix      : Char; {repeat prefix}
    CapabilitiesMask  : Byte; {extended capabilities}
    WindowSize        : Byte; {window size}
    MaxLongPacketLen  : Word; {maximum long packet}
  end;
```

This record contains the various options that Kermit can control. The default values and the meanings of the options are described in "Kermit Options" earlier in this section. You need to be concerned about a KermiTOptionRec only if you want to change the defaults used by Async Professional.

APKERMIT Declarations

Types

```
KermiTPtr = ^KermiTProtocol;
KermiTProtocol =
  record
    ...
  end;
```

Record used to store data needed by the various routines in APKERMIT to implement the Kermit protocol.

Procedures and Functions

```
procedure PrepareReceivePartKM(P : ProtocolRecPtr);
  {-Prepare to call the background function ProtocolReceivePartKM}

procedure PrepareTransmitPartKM(P : ProtocolRecPtr);
  {-Prepare to use the background function ProtocolTransmitPartKM}

procedure ProtocolReceiveKM(P : ProtocolRecPtr);
  {-Receive a batch of files transmitted using Kermit}

function ProtocolReceivePartKM(P : ProtocolRecPtr) : ProtocolStateType;
  {-Perform one increment of a background protocol receive}
```

```

procedure ProtocolTransmitKM(P : ProtocolRecPtr);
  {-Start Kermit protocol transmit}
function ProtocolTransmitPartKM(P : ProtocolRecPtr) : ProtocolStateType;
  {-Perform one increment of a background protocol transmit}

```

These functions are not documented here because they work exactly the same as the similarly named functions in APABSPCL (for example, PrepareReceivePartKM is documented in PrepareReceivePart). See §7.C for a complete discussion of these functions.

OOKERMIT Declarations

Types

```

KermitProtocolPtr = ^KermitProtocol;
KermitProtocol =
  object(AbstractProtocol)
    ...
  end;

```

Object that defines data fields and methods needed to implement the Kermit protocol.

Procedures and Functions

```

procedure KermitProtocol.PrepareReceivePart; virtual;
  {-Prepare to call the background function ProtocolReceivePart}
procedure KermitProtocol.PrepareTransmitPart; virtual;
  {-Prepare to use the background function ProtocolTransmitPart}
procedure KermitProtocol.ProtocolReceive; virtual;
  {-Receive a batch of files transmitted using Kermit}
function KermitProtocol.ProtocolReceivePart : ProtocolStateType; virtual;
  {-Perform one increment of a background protocol receive}
procedure KermitProtocol.ProtocolTransmit; virtual;
  {-Start Kermit protocol transmit}
function KermitProtocol.ProtocolTransmitPart : ProtocolStateType; virtual;
  {-Perform one increment of a background protocol transmit}

```

These functions are not documented here because they work exactly the same as the ones with the same name in OOABSPCL. See §7.C for a complete discussion.

Syntax

destructor KermitProtocol.Done; virtual;

Purpose

Dispose of the object.

Description

This destructor disposes of the internal data structures allocated on the heap by the Init constructor and destroys the object.

Example

See the example for InitCustom.

Syntax

procedure DoneKermit(var P : ProtocolRecPtr);

Purpose

Dispose of the protocol record.

Description

This routine disposes of the protocol record pointed to by P, previously allocated by a call to InitKermit, as well as the data structures that were allocated on the heap.

Example

See the examples for InitCustomKermit and InitKermit.

Syntax

```
procedure GetLPStatus(P : ProtocolRecPtr; var InUse : Boolean;
                     var PacketSize : Word);
procedure KermitProtocol.GetLPStatus(var InUse : Boolean; var PacketSize : Word);
```

Purpose

Return the state of long packet usage and the packet size.

Description

This procedure is primarily for use in your status routine to show if long packets are in use and the current long packet size. If InUse returns False, PacketSize contains the size of a standard packet, otherwise it contains the size for long packets. If this routine is called prior to negotiating the packet size, it returns the default value for long packet size; if called after handshaking, it returns the size actually in use.

Example

```
{SF+}
procedure MyStatusProc(P : ProtocolRecPtr; Starting, Ending : Boolean);
var
    InUse : Boolean;
    PacketSize : Word;
begin
    GetLPStatus(P, InUse, PacketSize);
    if InUse then
        WriteLn('Using long packets of size: ', PacketSize);
    end;
```

A simple status function that displays the current size of the long packets if they're being used. The OOP version of this function would not require the P parameter.

Syntax

```
function GetSwcSize(P : ProtocolRecPtr) : Byte;
function KermitProtocol.GetSwcSize : Byte;
```

Purpose

Return the number of slots in the Sliding Windows Control table.

Description

This function is designed primarily for use in status procedures. It returns the size of the sliding window table used during an SWC protocol session. If SWC is not enabled or was turned off by the protocol handshake, this function returns zero. If called before handshaking and SWC is enabled, it returns the default number of windows as set by SetMaxWindows. If called after handshaking, it returns the number of windows actually in use.

Example

```
var
    KM : ProtocolRecPtr;
...
InitKermit(KM, UP);
...
WriteLn('Sliding window size: ', GetSwcSize(KM));
```

Shows the sliding window size being used. This may be different than the size requested by Async Professional if the other side requests a smaller size. When using objects, the KM parameters would be omitted.

Syntax

```
constructor KermitProtocol.Init(APPtr : AbstractPortPtr);
```

Purpose

Allocate and initialize a protocol control block.

Description

This constructor initializes a KermitProtocol object using the default protocol options (DefProtocolOptions). See InitCustom for further details.

Example

```
var
  UP : UartPort;
  KP : KermitProtocol;
...
  UP.InitFast(Com1, 2400);
...
  if not KP.Init(@UP) then {handle error} ;
...
  KP.Done;
```

Set up for a Kermit file transfer at 2400 baud using Com1.

See Also

KermitProtocol.InitCustom

InitKermit

Syntax

```
constructor KermitProtocol.InitCustom(APPtr : AbstractPortPtr;
                                       KOptions : KermitOptionRec;
                                       Options : Word);
```

Purpose

Allocate and initialize a protocol control block with the specified options.

Description

This constructor initializes a KermitProtocol object for file transfers using the specified port, Kermit options (normally DefKermitOptions), and protocol Options (normally DefProtocolOptions). It also allocates space on the heap for some large data structures needed for internal use (9KB).

Example

```
var
  UP : UartPort;
  KP : KermitProtocol;
...
  UP.InitFast(Com1, 2400);
...
  if not KP.InitCustom(@UP, DefKermitOptions, DefProtocolOptions) then
    {handle error} ;
...
  KP.Done;
```

Set up for a Kermit file transfer at 2400 baud using Com1.

See Also

InitCustomKermit

SetKermitOptions

InitCustomKermit / InitKermit

Syntax

```
procedure InitCustomKermit(var P : ProtocolRecPtr; PortPtr : PortRecPtr;  
                           KOptions : KermitOptionRec; Options : Word);
```

Purpose

Allocate and initialize a protocol control block with the specified options.

Description

This routine initializes a protocol record for Kermit file transfers using the specified port, Kermit options (normally DefKermitOptions), and protocol Options (normally DefProtocolOptions). It also allocates space on the heap for some large data structures needed for internal use (9KB).

Example

```
var  
  KP : ProtocolRecPtr;  
  P : PortRecPtr;  
...  
  InitPortFast(P, Com1, 2400);  
...  
  InitKermit(KP, P, DefKermitOptions, DefProtocolOptions);  
...  
  DoneKermit(KP);
```

Set up for a Kermit file transfer at 2400 baud using Com1.

See Also

KermitProtocol.InitCustom

SetKermitOptions

Syntax

```
procedure InitKermit(var P : ProtocolRecPtr; PortPtr : PortRecPtr);
```

Purpose

Allocate and initialize a protocol control block.

Description

This routine initializes a protocol record for Kermit file transfers using the default protocol options (DefProtocolOptions) and default Kermit options (DefKermitOptions). See InitCustomKermit for further details.

Example

```
var  
  KP : ProtocolRecPtr;  
  P : PortRecPtr;  
...  
  InitPortFast(P, Com1, 2400);  
...  
  InitKermit(KP, P);  
...  
  DoneKermit(KP);
```

Set up for a Kermit file transfer at 2400 baud using Com1.

See Also

KermitProtocol.Init

InitCustomKermit

Syntax

```
{IFDEF UseStreams}
constructor KermitProtocol.Load(var S : IdStream);
```

Purpose

Load a KermitProtocol object from a stream.

Description

This constructor loads a KermitProtocol object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to KermitProtocol.Store.

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Load is called.

See the entry for XmodemProtocol.Store for a discussion of how to link a protocol object loaded from a stream with a port object.

The stream registration routine for a KermitProtocol object is KermitProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses
  OpRoot, ApPort, OoCom, OoAbsPcl, OoKermit;
var
  UP : UartPort;
  KP : KermitProtocol;
  S : BufIdStream;
  Status : Word;
begin
  if not S.Init('KERMIT.STM', SOpen, 1024) then begin
    WriteLn('Error opening stream');
    Halt;
  end;
  S.RegisterHier(KermitProtocolStream);
  S.RegisterHier(UartPortStream);
  S.RegisterPointer(1000, @UP);
  S.Get(UP);
  S.Get(KP);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Load error: ', Status);
    Halt;
  end;
  WriteLn('Loaded successfully');
  S.Done;
  KP.Done;
  UP.Done;
end.
```

Instantiates UP and KP from a stream created by the Store example using the stream's Get method, which calls the objects' Load constructors indirectly.

Notice that a pointer to the port object is registered to ensure that KermitProtocol.Load can create a link between the two objects.

SetCtlPrefix / SetHibitPrefix

Syntax

```
procedure SetCtlPrefix(P : ProtocolRecPtr; C : Char);  
procedure KermitProtocol.SetCtlPrefix(C : Char);
```

Purpose

Set the control character quote prefix.

Description

This procedure changes the prefix character Kermit uses when quoting control characters. Kermit uses the '#' character by default. C is the new prefix character to use.

SetCtlPrefix should only be called before a transfer starts.

Example

```
var KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetCtlPrefix(KM, '$');
```

Changes the control character prefix to '\$'. When using objects, the KM parameters would be omitted.

See Also

SetKermitOptions

Syntax

```
procedure SetHibitPrefix(P : ProtocolRecPtr; C : Char);  
procedure KermitProtocol.SetHibitPrefix(C : Char);
```

Purpose

Set the high bit quote prefix.

Description

This procedure changes the prefix character used when quoting data bytes that have the high bit (bit 7) set.

The value specified in C has some special meanings. If C equals 'Y', this implementation of Kermit can use high bit prefixing if required by the remote, but it doesn't need this feature itself. The prefix character will be specified by the remote, if the remote requires high bit prefixing.

If C equals '&' or is in the range 33 - 62 or 96-126, this implementation of Kermit *requires* high bit prefixing. The character in C will be used as the prefix character.

If C equals 'N' or any value not listed here, this implementation of Kermit won't (or can't) use high bit prefixing.

Async Professional Kermit sets this option to 'Y' by default because it doesn't *require* that high bit prefixing be used. However, if the remote computer asks for prefixing, it will be honored.

Example

```
var KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetHibitPrefix(KM, '&');
```

Forces high bit prefixing. The prefix character will be '&'. When using objects, the KM parameters would be omitted.

See Also

SetKermitOptions

Syntax

```
procedure SetKermitCheck(P : ProtocolRecPtr; CType : Byte);  
procedure KermitProtocol.SetKermitCheck(CType : Byte);
```

Purpose

Set the block check type.

Description

This procedure sets the block check method used to verify the integrity of each packet. Kermit uses a one-byte checksum by default (bcChecksum1).

CType is the new block check method to use. The only acceptable values are:

bcChecksum1	- a one-byte checksum
bcChecksum2	- a two-byte checksum
bcCrck	- a three-byte CRC

Any other value generates an `ecInvalidArgument` error (and `bcChecksum1` is used by default).

Examples

```
var  
  KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetKermitCheck(KM, bcCrck);
```

Changes the block check method from the default one-byte checksum to a three-byte CRC. When using objects, the KM parameters would be omitted.

```
SetKermitCheck(KM, bcCrck32);
```

Incorrect. Kermit does not support 32 bit CRCs. This call returns an `ecInvalidArgument` error.

See Also

SetKermitOptions

SetKermitOptions

Syntax

```
procedure SetKermitOptions(P : ProtocolRecPtr; KOptions : KermitOptionRec);  
procedure KermitProtocol.SetKermitOptions(KOptions : KermitOptionRec);
```

Purpose

Update the protocol to use new Kermit options.

Description

This routine modifies some or all of the various Kermit options by supplying a complete new options record. See "Kermit Options" earlier in this section for a discussion of the various options.

Use this method when you need to change all or most of the Kermit options. When you need to change only one or two options, you should use the specific option routines listed below.

Example

```
const  
  MyKermitOptions : KermitOptionRec =  
    (MaxPacketLen : 90;           {90 characters}  
     MaxTimeout : 10;            {10 seconds}  
     PadCount : 0;               {No pad chars}  
     PadChar : #0;               {Null pad char}  
     Terminator : cCR;           {Carriage return}  
     CtlPrefix : '#';           {'#' char}  
     HbitPrefix : 'Y';          {Can handle hbit prefixing}  
     Check : '1';                {1 byte chksum}  
     RepeatPrefix : '~';         {Default repeat prefix}  
     CapabilitiesMask : 0;        {No default extended caps}  
     WindowSize : 0;             {No default windows}  
     MaxLongPacketLen : 0);      {No default long packets}  
  
var  
  UP : UartPortPtr;  
  KM : ProtocolRecPtr;  
  ...  
  InitKermit(KM, UP);  
  SetKermitOptions(KM, MyKermitOptions);
```

MyKermitOptions uses a default maximum packet length of 90 (instead of 80) and maximum packet timeout of 10 seconds (instead of 5). Note that MyKermitOptions could also have been supplied when the protocol was initialized (via InitCustom or InitCustomKermit).

See Also

KermitProtocol.InitCustom	SetMaxPacketLen
InitCustomKermit	SetMaxTimeoutSecs
SetCtlPrefix	SetMaxWindows
SetHbitPrefix	SetPacketPadding
SetKermitCheck	SetRepeatPrefix
SetMaxLongPacketLen	SetTerminator

SetMaxLongPacketLen / SetMaxPacketLen

Syntax

```
procedure SetMaxLongPacketLen(P : ProtocolRecPtr; MaxLen : Word);  
procedure KermitProtocol.SetMaxLongPacketLen(MaxLen : Word);
```

Purpose

Turn long packet support on or off.

Description

Passing a value of 0 in MaxLen turns off long packets, while passing a value between 1 and 1024 sets that value as the maximum length for long packets. Passing a value higher than 1024 returns an error (epNonFatal+ecInvalidArgument). This procedure sets or clears the apKermitLongPackets flag as appropriate.

This procedure must be called after you initialize the KermitProtocol object or ProtocolRecPtr, but before starting a protocol transfer session; calling it after a protocol session starts will lead to unpredictable results.

Example

See SIMPCOM.PAS for an example of long packet use.

Syntax

```
procedure SetMaxPacketLen(P : ProtocolRecPtr; MaxLen : Byte);  
procedure KermitProtocol.SetMaxPacketLen(MaxLen : Byte);
```

Purpose

Set the maximum packet length.

Description

This procedure sets the maximum number of bytes transferred in one packet. The default is 80.

MaxLen is the new desired maximum packet length. The maximum acceptable value is 94.

Values higher than that will fail with an ecInvalidArgument error.

This procedure must be called after you initialize the KermitProtocol object or ProtocolRecPtr, but before starting a protocol transfer session; calling it after a protocol session starts will lead to unpredictable results.

Example

```
var  
  KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetMaxPacketLen(KM, 90);
```

Changes the maximum packet length from 80 (the default) to 90. When using objects, the KM parameters would be omitted.

See Also

SetKermitOptions

SetMaxTimeoutSecs / SetMaxWindows

Syntax

```
procedure SetMaxTimeoutSecs(P : ProtocolRecPtr; MaxTimeout : Byte);  
procedure KermitProtocol.SetMaxTimeoutSecs(MaxTimeout : Byte);
```

Purpose

Set the maximum time to wait for a packet (in seconds).

Description

This procedure changes the maximum time the protocol will wait to receive a packet. If a packet is not received within this time period, the protocol is aborted.

MaxTimeout contains the new timeout value in seconds. Note that the unit of measurement is different from most timeout parameters in Async Professional (which are expressed in clock ticks).

This procedure must be called after you initialize the KermitProtocol object or ProtocolRecPtr, but before starting a protocol transfer session; calling it after a protocol session starts will lead to unpredictable results.

Example

```
var  
  KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetMaxTimeoutSecs(KM, 10);
```

Changes the maximum packet timeout from 5 seconds (the default) to 10 seconds. When using objects, the KM parameters would be omitted.

See Also

SetKermitOptions

Syntax

```
procedure SetMaxWindows(P : ProtocolRecPtr; MaxNum : Byte);  
procedure KermitProtocol.SetMaxWindows(MaxNum : Byte);
```

Purpose

Turn Sliding Windows Control (SWC) on or off.

Description

If MaxNum is 0, SWC is turned off. If MaxNum is between 1 and MaxWindowSlots, the maximum number of table slots ("window size") is set to MaxNum. If MaxNum is larger than MaxWindowSlots, an error (epNonFatal+ecInvalidArgument) is returned. This function sets or clears the apKermitSWC flag as appropriate.

If the two sides of a Kermit transfer specify different window sizes, the smaller of the two sizes is used.

This procedure deallocates the existing buffer and allocates a new buffer. The size of the new buffer is MaxNum*MaxLen. MaxLen is the value passed to the last call to SetMaxPacketLen or SetMaxLongPacketLen.

SetMaxWindows must be called after you initialize the KermitProtocol object or ProtocolRecPtr, but before starting a protocol transfer session; calling it after a protocol session starts will lead to unpredictable results.

Example

```
var  
  KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetMaxWindows(KM, 15);
```

Changes the window size to 15 slots. When using objects, the KM parameters would be omitted.

Syntax

```
procedure SetPacketPadding(P : ProtocolRecPtr; C : Char; Count : Byte);  
procedure KermitProtocol.SetPacketPadding(C : Char; Count : Byte);
```

Purpose

Set the packet pad character and pad count.

Description

This procedure changes the packet padding character and the number of padding characters to supply. Padding characters are added to the front of every Kermit packet, usually to account for a remote computer that is slow in switching from sending mode to receiving mode. By default, Kermit doesn't send any padding characters.

C is the pad character to send and Count is the number of pad characters to send.

This procedure must be called after you initialize the KermitProtocol object or ProtocolRecPtr, but before starting a protocol transfer session; calling it after a protocol session starts will lead to unpredictable results.

Example

```
var  
  KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetPacketPadding(KM, ' ', 10);
```

Changes KM to send 10 space characters before the start of each packet. When using objects, the KM parameters would be omitted.

See Also

SetKermitOptions

Syntax

```
procedure SetRepeatPrefix(P : ProtocolRecPtr; C : Char);  
procedure KermitProtocol.SetRepeatPrefix(C : Char);
```

Purpose

Set the repeat quote prefix character.

Description

This procedure changes the prefix character used when Kermit compresses repeated strings of characters. Kermit uses the '^' character by default. C is the new repeat prefix character to use.

SetRepeatPrefix should only be called before a transfer starts.

This procedure must be called after you initialize the KermitProtocol object or ProtocolRecPtr, but before starting a protocol transfer session; calling it after a protocol session starts will lead to unpredictable results.

Example

```
var  
  KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetRepeatPrefix(KM, '*');
```

Changes the repeat character prefix to '*'. When using objects, the KM parameters would be omitted.

See Also

SetKermitOptions

SetSWCTurnDelay / SetTerminator

Syntax

```
procedure SetSWCTurnDelay(P : ProtocolRecPtr; TrnDelay : Word);  
procedure KermitProtocol.SetSWCTurnDelay(TrnDelay : Word);
```

Purpose

Adjust the turnaround delay factor used when SWC is enabled.

Description

The protocol's internal TurnDelay value is set to TrnDelay ticks (the default is 5 ticks) when sliding windows are used. If sliding windows are not used, then the protocol uses its normal TurnDelay value.

Example

```
var  
  KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetSWCTurnDelay(KM, 100);
```

Sets the turn delay to 100 ticks if sliding windows are used. When using objects, the KM parameters would be omitted.

Syntax

```
procedure SetTerminator(P : ProtocolRecPtr; C : Char);  
procedure KermitProtocol.SetTerminator(C : Char);
```

Purpose

Set the packet terminator.

Description

This procedure changes the packet terminator, which is a carriage return by default. C is the new terminator to use.

You should not need to change the terminator character. It's required only by systems that need an end-of-line character before they can start processing.

This procedure must be called after you initialize the KermitProtocol object or ProtocolRecPtr, but before starting a protocol transfer session; calling it after a protocol session starts will lead to unpredictable results.

Example

```
var  
  KM : ProtocolRecPtr;  
...  
  InitKermit(KM, UP);  
  SetTerminator(KM, cLF);
```

Changes the terminator character to a line feed. When using objects, the KM parameters would be omitted.

See Also

SetKermitOptions

Syntax

```
($IFDEF UseStreams)
procedure KermitProtocol.Store(var S : IdStream);
```

Purpose

Store a KermitProtocol object to a stream.

Description

Store writes an image of the KermitProtocol object to the stream S.

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Store is called.

More importantly, you must give KermitProtocol.Store some means of forming a link between the protocol object and the port object whose address was passed to the Init constructor. See the entry for XmodemProtocol.Store for a discussion of this issue.

The stream registration routine for a KermitProtocol object is KermitProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses
  OpRoot, ApPort, OoCom, OoAbsPcl, OoKermit;
var
  UP : UartPort;
  KP : KermitProtocol;
  S : BufIdStream;
  Status : Word;
begin
  if not UP.InitFast(Com1, 2400) then begin
    WriteLn('Error initializing port');
    Halt;
  end;
  if not KP.Init(@UP) then begin
    WriteLn('Error initializing protocol');
    Halt;
  end;
  if not S.Init('KERMIT.STM', SCreate, 1024) then begin
    WriteLn('Error initializing stream');
    Halt;
  end;
  S.RegisterHier(KermitProtocolStream);
  S.RegisterHier(UartPortStream);
  S.RegisterPointer(1000, @UP);
  S.Put(UP);
  S.Put(KP);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error: ', Status);
    Halt;
  end;
  WriteLn('Stored successfully');
  S.Done;
  KP.Done;
  UP.Done;
end.
```

Store

Instantiates the port and protocol objects and then stores them in a stream using the stream's Put method, which calls the object's Store method indirectly.

Notice that a pointer to the UartPort object is registered so that KermitProtocol.Store can create a link between the two objects.

See KermitProtocol.Load for an example program that reloads these objects.

See Also

KermitProtocol.Load

XmodemProtocol.Store

Syntax

```
function WindowsUsed(P : ProtocolRecPtr) : Word;  
function KermitProtocol.WindowsUsed : Word;
```

Purpose

Return the number of window slots used.

Description

This function is primarily for use in your status routine to display the number of window slots currently occupied. "Occupied" in this context means that a packet was transmitted (placed in the window) and Kermit is currently waiting for a response. For example, if 10 packets were transmitted, but no responses have been received, WindowsUsed returns the value 10. When an acknowledgement is received for the first packet, WindowsUsed returns the value 9.

In practice you will likely see WindowsUsed increase quickly to the maximum window value as the transmitted packets fill up the window buffers. It will not often drop below this until the file is completely transmitted.

Example

See SIMPCOM.PAS for an example of Sliding Windows use.

H. CompuServe B+ Protocol: APBPLUS/OOBPLUS

APBPLUS/OOBPLUS build on APABSPCL/OOABSPCL to provide the CompuServe B+ protocol. The B+ protocol is a proprietary file transfer and transport protocol designed and used exclusively by CompuServe.

The "B" protocol is offered by CompuServe in three varieties: classic B, Quick B, and B+. Async Professional supports only B+ since it has the most capabilities and is recommended by CompuServe. CompuServe maintains the older versions only for compatibility with older remote communications programs.

As with other file transfer protocols, the internal operations of B+ are not documented here since you simply don't need that level of detail to successfully use the protocol as provided by APBPLUS/OOBPLUS. If you need more detail, see BPLUS.LZH. It contains the official BPLUS.DOC documentation file provided by CompuServe.

B+ is a modern protocol providing many of the same features as Zmodem and Kermit. It is a block oriented protocol where each transmitted block requires an acknowledgement from the receiver. It does, however, offer a transmit send ahead feature (similar to Kermit's sliding windows) to minimize the time wasted waiting for acknowledgements. B+ currently supports a window size of two pending acknowledgements.

B+ also provides data link escaping for transparent operation of a variety of network types. Data link escaping, as with Zmodem and Kermit, means that control characters appearing in the datastream are modified before being transmitted. The receiver restores them to their original state. Like Kermit, B+ refers to this process as "quoting." And, like Zmodem, B+ is capable of resuming interrupted file transfers. B+ can resume interrupted receive file sessions only; interrupted transmit file sessions must always be restarted from the beginning.

Of all the protocols supported by Async Professional, B+ is the most unusual. All of the other protocols are true file transfer protocols. B+ can be a transport layer protocol as well as a file transfer protocol. This means it can provide error correction and guaranteed reception features to all parts of CompuServe. Async Professional's implementation of B+ does not attempt to provide these transport layer features. However, the supplied bonus unit GIFDEMO demonstrates one way of using APBPLUS/OOBPLUS as a transport layer. It receives and displays GIF graphics using B+ as a transport layer protocol.

B+ differs from all other supported protocols in another way as well: your program is always the slave and CompuServe is always the host. The CompuServe host always starts and controls the protocol whether you are transmitting or receiving files. For this reason, you can't just call ProtocolTransmit or ProtocolReceive when you want to. You must wait until CompuServe tells you it's ready to start a protocol. It also tells you whether to receive or transmit and the name of the file.

To handle this properly, your terminal loop must constantly be on the lookout for an indication from CompuServe that it wants to start a file transfer. Worse yet, several protocol control packets are exchanged while you are still in your terminal loop. The B+ specification requires that you remain in your terminal loop while processing these packets since non-protocol data can arrive between the packets (non-protocol data meaning data that your terminal loop should display, including prompts that may require a response from the user). Only after processing these packets can you call ProtocolTransmit or ProtocolReceive to process the rest (and the majority of) the file transfer.

Before looking at this process in detail, here is the basic flow of packets in a B+ protocol, where your terminal is receiving a file from a CompuServe host:

CompuServe	Remote	Explanation
<ENQ>	---->	initialize your B+ variables
	<--- <DLE>++<DLE>0	send acknowledge
<DLE>B<+ packet>	---->	CompuServe's info packet
	<--- <DLE>B<+packet>	your info packet
<DLE>#	---->	receive acknowledge
<DLE>B<T packet>	---->	receive file info packet
	<--- <DLE>#	send acknowledge

Your program must remain in its terminal loop until all of the above packets are exchanged because other data (not related to the protocol) could arrive during this time. Once you've received the T packet, however, you can call ProtocolTransmit or ProtocolReceive, because from now on you will only receive protocol data. The protocol continues as shown below:

CompuServe	Remote	Explanation
<DLE>B<N packet>	---->	receive next data packet
	<--- <DLE>#	send acknowledge
<DLE>B<N packet>	---->	receive next data packet
	<--- <DLE>#	send acknowledge
...		
<DLE>B<TC packet>	---->	receive transfer complete packet
	<--- <DLE>#	send acknowledge

The TC (transfer complete) packet signals that all of the file data was transmitted and the protocol is over. B+ is not capable of sending multiple files, so ProtocolReceive and ProtocolTransmit return control to your program at this point.

Quoted Data

B+ "quotes" certain control characters. Quoting means that these control characters are modified by the transmitter and restored by the receiver. The result is that the datastream *never* contains these control characters. This allows B+ to run over networks that use control characters for flow control or other network operations.

Quoting is handling automatically by APBPLUS/OOBPLUS; there's nothing you need to do to turn it on or off. The following characters are quoted by default:

```
<ETX>    $03
<ENQ>    $05
<DLE>    $10
<DC1>    $11
<DC3>    $13
<NAK>    $15
```

The B+ protocol provides a mechanism by which your program can request that a different set of characters be quoted. Because it is highly unlikely you'll ever need to do this, APBPLUS/OOBPLUS don't provide any methods or functions to support it. However, if you ever do need to provide your own quote set, the simplest way to do it is to modify the typed constant DQDefault:

```
type
  QuoteArray = array[0..7] of Byte;
const
  DQDefault : QuoteArray = ($14, $00, $D4, $00, $00, $00, $00, $00);
```

The first four bytes of this array represent the ASCII codes \$00-\$1F and the last four bytes represent ASCII codes \$80-\$9F. If a particular bit is on, then that ASCII code is to be quoted.

B+ Terminal Processing

As noted earlier, use of the B+ protocol requires participation of your terminal loop. Since CompuServe tells you when it is time to start the protocol and whether you are to receive a file or transmit a file, your terminal loop must constantly check for certain handshaking characters provided by CompuServe.

The first such character is <ENQ>. CompuServe calls this the Enquire Control Sequence, even though it's just one character. It is sent by CompuServe as a signal for the slave (your program) to prepare for the B+ protocol. (It's also used internally to resynchronize after an error, but that use is not of concern here.)

According to the B+ specification, when your program receives the <ENQ> character, it should take whatever steps are necessary for it to enter into a B+ protocol session and send a special acknowledge string. It should *not* start the session, though. With APBPLUS/OOBPLUS, this is handled by the ProcessENQ method. So, the first modification you must make to your terminal loop is code to handle the <ENQ>:

```
{...init the port object ComPort...}
{...init the B+ object BP...}

repeat
  {Handle received data}
  if ComPort.CharReady then begin
    ComPort.GetChar(C);
    case C of
      cENQ : BP.ProcessENQ;           {<----- required by B+}
      else  Write(C);
    end;
  end;
until Finished;

{...handle keyboard data...}
```

The <ENQ> can arrive at any time. It is typically sent by CompuServe after the user interactively requests a B+ upload or download. However, it can be sent at other times also. The most notable other time is immediately after logon. CompuServe waits 10 seconds for a response to the <ENQ>. If you don't call BP.ProcessENQ in response to the <ENQ>, your terminal session will suffer from these 10 second delays.

This creates somewhat of a dilemma for general purpose programs—programs that are used for purposes other than CompuServe access. Obviously the BP protocol object must be instantiated before it can process the <ENQ>. You have several ways to accomplish this:

1. Instantiate the protocol object when your program starts up so that it is always ready.
2. Have the user flag the session as a CompuServe session and only create the protocol object when that flag is set.
3. Instantiate the protocol object the first time you receive an <ENQ> before calling BP.ProcessENQ.

All of these approaches will work; which you choose depends on application requirements such as memory availability and how often CompuServe is used.

The second character your terminal loop must check for is the <DLE> character. The <DLE> character followed by a 'B' (<DLE>B) is called the Lead-in Sequence. Every B+ packet starts with the Lead-in Sequence. While in terminal mode (not in B+ mode) CompuServe never sends out a

<DLE> unless it is starting a B+ packet. So, when your terminal loop sees a <DLE>, it's safe to assume that a B+ packet is being started by CompuServe.

According to the B+ specification, when you receive a <DLE> character, you should receive the rest of the packet and process it according to packet type. APBPLUS/OOBPLUS accomplish this through the ProcessDLE function. There are two types of packets that may be received while in terminal mode: the '+' packet with protocol option information and the 'T' packet with file information. When ProcessDLE finds a '+' packet, it notes the protocol options requested by CompuServe and sends a '+' packet with the APBPLUS/OOBPLUS options. Control then returns to your terminal loop with ProcessDLE's var parameter Start set to False—meaning that your terminal loop should *not* start the protocol yet.

When ProcessDLE finds a 'T' packet, it notes the file name and whether CompuServe is asking for an upload or download. It then returns control to your terminal loop. In this case, however, it sets the var parameter Start to True—meaning that your terminal loop should now start the protocol by calling either ProtocolTransmit or ProtocolReceive. Which of these two you should call depends on ProcessDLE's second var parameter, Upload. When Upload is True, you call ProtocolTransmit; when Upload is False, you call ProtocolReceive. You do *not* need to call SetFileMask before calling ProtocolTransmit. In fact, ProtocolTransmit does not use the abstract protocol NextFile hook to get the name of the file to transmit. Instead, it relies on CompuServe to tell it what file to transmit. (Note that APBPLUS/OOBPLUS are the *only* Async Professional protocols with this behavior. All other protocols use the NextFile hook to get the name of the file to transmit.)

So, another modification to the terminal loop is necessary:

```
{...init the port object ComPort...}
{...init the B+ object BP...}

repeat
  {Handle received data}
  if ComPort.CharReady then begin
    ComPort.GetChar(C);
    case C of
      cENQ : BP.ProcessENQ;           {<----- required by B+>}
      cDLE : begin                    {<----- required by B+>}
        BP.ProcessDLE(Start, Upload); {<----- required by B+>}
        if Start then begin          {<----- required by B+>}
          if Upload then              {<----- required by B+>}
            BP.ProtocolTransmit       {<----- required by B+>}
          else                        {<----- required by B+>}
            BP.ProtocolReceive;       {<----- required by B+>}
          end;                        {<----- required by B+>}
        end;                          {<----- required by B+>}
      else Write(C);
    end;
  end;

  {...handle keyboard data...}
until Finished;
```

Once you call ProtocolTransmit or ProtocolReceive, ABPLUS/OOBPLUS behave like any Async Professional protocol: they periodically call your status routine, they can be aborted via the port abort hook, and so on. They can also be run in the background by calling ProtocolTransmitPart or ProtocolReceivePart at intervals that you choose.

ProcessDLE is *not* a background routine. It maintains control until the entire packet is received or a timeout error occurs. The protocol doesn't enter its "background" mode until you call ProtocolTransmit or ProtocolReceive.

The third character sequence your terminal loop must check for is <ESC>I. If your CompuServe profile specifies VT52 terminal emulation, your program may receive the sequence <ESC>I, called the CompuServe Interrogation Sequence. Your program should return an information string telling CompuServe about your terminal capabilities. See ProcessESCI for more information.

B+ Terminal Processing using a TerminalWindow

B+ processing when using a TerminalWindow requires that you handle all of the above issues, but in a completely different fashion. When using a TerminalWindow, your program doesn't have a terminal loop. Instead, TerminalWindow itself has the terminal loop and it handles all incoming data and keyboard data. You need to find the right place to "plug in" to the TerminalWindow to examine incoming data.

There are actually several approaches you could take: overriding ProcessSelf, overriding GetNextCommand, or overriding GetIncomingChar. If your program does exclusively B+ protocols, then you may want to consider overriding ProcessSelf and handling everything there. In most cases, however, overriding GetNextCommand is probably the simplest and most straightforward approach.

The supplied example handles the <ENQ>, <DLE>, and <ESC>I sequences *within* GetNextCommand since these are fairly fast and simple actions. When ProcessDLE returns Start=True, meaning the protocol should be started, an exit command is set to force TerminalWindow.Process to exit back to your application. Then, it's simply a matter of adding your chosen exit commands to your existing GetLastCommand case statement. This makes the code for processing B+ protocols as similar to other protocols as possible.

Here are some excerpts (from the example program EXBPLUS) that show how your derived terminal window object might look:

```
type
  BPlusTermWin = object(TerminalWindow)
    Started : Boolean;
    function StartBPlus : Boolean;
    procedure GetNextCommand; virtual;
  end;
```

The field Started and the method StartBPlus help to decide when to instantiate the BPlus protocol object. This is necessary unless you want to instantiate the BPlus object at program startup time and leave it for the duration of the application. That is, your terminal window can receive an <ENQ> or <DLE> at *any* time. So, you either need to instantiate the protocol as soon as you see one of these characters, or the protocol must already be instantiated. This example does not instantiate until a character that requires it is received:

```
function BPlusTermWin.StartBPlus : Boolean;
{-Start BPlus}
begin
  if not Started then begin
    BP.Init(@UP);
    BP.SetShowStatusProc(ShowStatus);
  end;
  Started := AsyncStatus = ecOk;
  StartBPlus := Started;
end;
```

The next step is to examine each character received by the terminal window. Overriding `GetNextCommand` provides a simple way to do this. When `GetIncomingChar` returns `True`, this character is checked against `<ENQ>`, `<DLE>`, and the sequence `<ESC>I`. If any of these characters are found, the appropriate `B+` method is called to handle it, starting the protocol as it does so.

```

procedure BPlusTermWin.GetNextCommand;
const
  Index : Byte = 0;
var
  C : Char;
  Start, IsUpload : Boolean;
begin
  while not cwCmdPtr^.cpKeyPressed do
    if GetIncomingChar(C) then begin
      case C of
        cENQ : if StartBPlus then
          BP.ProcessENQ;
        cDLE : if StartBPlus then begin
          BP.ProcessDLE(Start, IsUpload);
          if Start then begin
            if IsUpload then
              cwCmd := ccUser1
            else
              cwCmd := ccUser2;
            Exit;
          end;
        end;
      else begin
        if CheckForString(Index, C, cEsc+'I', True) then begin
          BP.ProcessEscI(80, 25);
          AnsiEmulatorPtr(twEmulator)`.InitParser;
        end else begin
          cwCmd := ccIncomingChar;
          cwKey := Word(C);
          Exit;
        end;
      end;
    end;
  end;
  end;
  end;
  TerminalWindow.GetNextCommand;
end;

```

If `ProcessDLE` indicates it's time to start the protocol, then an appropriate exit command is set (`ccUser1` for upload and `ccUser2` for download, in this example). This causes `TerminalWindow.Process` to exit back to your application and let the application handle the call to `ProtocolTransmit` or `ProtocolReceive`:

```

repeat
  TW.Process;
  case TW.GetLastCommand of
    ccUser1 : begin
      BP.ProtocolTransmit;
      BP.Done;
      TW.Started := False;
    end;
    ccUser2 : begin
      BP.ProtocolReceive;
      BP.Done;
    end;
  end;
until TW.Process = 0;

```

```

        TW.Started := False;
    end;
ccUser3..ccUser255
begin
    ...handle other exit commands...
end;
ccQuit, ccError : Finished := True;
end;
until Finished;

```

Resuming Interrupted Transfers

B+ is capable of resuming interrupted receive file sessions. It cannot resume interrupted transmit file sessions since CompuServe always discards partially transmitted files. The B+ mechanism for resuming files is more robust than Zmodem's standard method. Before resuming a file transfer, the slave (your program) calculates a CRC on the part of the file that it already has and transmits that value. CompuServe calculates a CRC on that same portion of the file. If the values match, the file transfer starts from the interrupted point. If the values don't match, the file transfer starts from the beginning of the file.

Although the above processing is automatic, you have control over whether it is performed or not. It is controlled by the abstract protocol option for handling name collisions. The choices, from APABSPCL/OOABSPCL are:

```
WriteFailOptions = (WriteFail, WriteRename, WriteAnyway, WriteResume);
```

and are set by a call to `SetOverwriteOption`. The default option is `WriteFail`, meaning that if an incoming file has the same name as an existing file, that file will be refused.

To take advantage of the B+ file resume feature, call `SetOverwriteOption(WriteResume)` meaning that APBPLUS/OOBPLUS will attempt to resume a file transfer when a name collision is found. That's all it takes; APBPLUS/OOBPLUS handle the resume automatically from then on.

Suppose, however, that you want to control resumes on a case by case basis—you don't want to resume a file until you see the file name. Or, you may want to let the user decide whether or not to resume an incoming file. APBPLUS/OOBPLUS provide for this via a B+ user hook called the "handle resume hook." This is a function that you supply and that APBPLUS/OOBPLUS call whenever they detect a name collision. This is your opportunity to set or change the `WriteFail` option. If you supply such a function, it's usually because you want the user to decide. So, your handle resume function would simply prompt the user for the desired action (usually fail, overwrite, rename, or resume). It would then call `SetOverwriteOption` as required.

Here's an example handle resume procedure:

```

{$F+}
procedure MyHandleResume(AP : AbstractProtocolPtr);
begin
    with AP^ do begin
        if Pathname = 'AUTOEXEC.BAT' then
            SetOverwriteOption(WriteFail)
        else
            SetOverwriteOption(WriteResume);
        end;
    end;
end;

```

This rather simple example just checks the incoming file name and attempts to resume unless the file name is 'AUTOEXEC.BAT'.

This example shows the OOP calling format. The non-OOP format is nearly identical, replacing the AbstractProtocolPtr with a ProtocolRecPtr. Here's the declaration of a non-OOP handle resume procedure:

```
procedure MyHandleResume(P : ProtocolRecPtr);
```

Other than this change, the OOP and non-OOP routines function exactly the same way.

A valid handle resume procedure must be declared FAR using {F+} or the far keyword, it must not be nested inside any other procedure, and it must take exactly the parameters shown in the examples above.

The default handle resume function is a built-in function called NoHandleResume. When this function is in place, the resume action is handled entirely by the existing value of the WriteFail option.

Logging on to CompuServe

This section has nothing to do with B+ protocol. However, if you are using the B+ protocol then you are most likely using CompuServe, since B+ is a proprietary CompuServe protocol. And logging on to CompuServe has a peculiarity that may confuse you if you're not aware of it.

All CompuServe logons, whether going through the CompuServe network or some other network service, start out at 7 data bits, even parity, and one stop bit. Most CompuServe users have their terminal profiles set to 8 data bits, no parity. This terminal profile becomes active *after* the logon process is complete.

There are two ways you can handle this. You can set your program to 7 data bits, even parity until the logon is complete, then switch to 8 data bits, no parity. Or, you can start at 8 data bits, no parity and simply mask off the high order bit of all characters before displaying them.

The disadvantage to the first approach is that it is possible to lose incoming data while making the switch. You can eliminate this possibility, though, by assuring that you make the switch while CompuServe is waiting for input from you (e.g., at a ! prompt).

The disadvantage to the second approach is that it may prevent some of the string checking functions of Async Professional (e.g., CheckForString and WaitForString) from working. The problem is that those functions handle all of the input themselves, without giving you a chance to mask off the high order bit. If you tell CheckForString to check for 'ABC', it may never receive that string if CompuServe transmits the 'B' character with the high bit (parity bit) set. You can solve this problem by deriving your own port object and overriding GetChar to strip the high bit. If you take this approach, be sure *not* to strip the high bit during protocol transfers; create your own flag to check for this or use the ProtocolInProgress function of the port object.

Shared Declarations

Constants

```
BPlusOverHead : Word = 20;
```

This is an estimate of the average number of overhead bytes per packet (required by EstimateTransferSecs to estimate the time required to transfer a specified number of bytes).

```
BPlusTurnDelay : Word = 0;
```

This is an estimate of the turnaround delay in a B+ protocol (required by EstimateTransferSecs to estimate the time required to transfer a specified number of bytes). Turnaround delay is the time, in milliseconds, required for the remote system to acknowledge the last block.

DDDefault : QuoteArray = (\$14, \$00, \$D4, \$00, \$00, \$00, \$00, \$00);
 The default QuoteArray used by B+. This quoting set should be acceptable for all of your B+ file transfers, however, you can change it if necessary.

ESCIResponse : String[80] = '#IB1,SSxx,GF,PB,DT';
 String transmitted by ProcessESCI in response to a received <ESC>I. xx is replaced by the current screen width and height.

Types

QuoteArray = Array[0..7] of Byte;
 A bitmap of character values that must be quoted before being transmitted. The first four bytes represent the ASCII codes \$00-\$1F. The second four bytes represent the ASCII codes \$80-\$9F.

APBPLUS Declarations

Types

```
BPlusPtr = ^BPlusProtocol;
BPlusProtocol =
  record
    ...
  end;
```

Record and record pointer used to store data needed to implement the B+ protocol.

HandleResumeProc = procedure(P : ProtocolRecPtr);

A procedure called by B+ when an incoming file has the same name as an existing file.

Procedures and Functions

```
procedure PrepareReceivePartBP(P : ProtocolRecPtr);
  {-Prepare to call the background function ProtocolReceivePartBP}
procedure PrepareTransmitPartBP(P : ProtocolRecPtr);
  {-Prepare to use the background function ProtocolTransmitPartBP}
procedure ProtocolReceiveBP(P : ProtocolRecPtr);
  {-Receive a batch of files transmitted using the B+ protocol}
function ProtocolReceivePartBP(P : ProtocolRecPtr) : ProtocolStateType;
  {-Perform one increment of a background protocol receive}
procedure ProtocolTransmitBP(P : ProtocolRecPtr);
  {-Start B+ protocol transmit}
function ProtocolTransmitPartBP(P : ProtocolRecPtr) : ProtocolStateType;
  {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the similarly named functions in APABSPCL (for example, PrepareReceivePartBP is documented in PrepareReceivePart). See §7.C for a complete discussion of these functions.

OOBPLUS Declarations

Types

```
BPlusProtocolPtr = ^BPlusProtocol;
BPlusProtocol =
  object(AbstractProtocol)
    ...
  end;
```

Object that defines data fields and methods needed to implement the B+ protocol.

HandleResumeProc = procedure(AP : BPlusProtocolPtr);

A procedure called by B+ when an incoming file has the same name as an existing file.

Procedures and Functions

```
procedure BPlusProtocol.PrepareReceivePart; virtual;  
    {-Prepare to call the background function ProtocolReceivePart}  
procedure BPlusProtocol.PrepareTransmitPart; virtual;  
    {-Prepare to use the background function ProtocolTransmitPart}  
procedure BPlusProtocol.ProtocolReceive; virtual;  
    {-Receive a batch of files transmitted using the B+ protocol}  
function BPlusProtocol.ProtocolReceivePart : ProtocolStateType; virtual;  
    {-Perform one increment of a background protocol receive}  
procedure BPlusProtocol.ProtocolTransmit; virtual;  
    {-Start B+ protocol transmit}  
function BPlusProtocol.ProtocolTransmitPart : ProtocolStateType; virtual;  
    {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the ones with the same name in OOABSPCL. See §7.C for a complete discussion.

Done / DoneBPlus / Init

Syntax

destructor BPlusProtocol.Done; virtual;

Purpose

Dispose of the protocol record.

Description

This destructor disposes of the internal data structures allocated on the heap by the Init constructor and destroys the object.

Example

See the example for BPlusProtocol.Init and BPlusProtocol.InitCustom.

Syntax

procedure DoneBPlus(var P : ProtocolRecPtr);

Purpose

Dispose of a B+ protocol record.

Description

This routine disposes of the protocol record pointed to by P and deallocates any memory owned by P. P must have been previously initialized by a call to InitBPlus or InitCustomBPlus.

Example

See the examples for InitBPlus and InitCustomBPlus.

See Also

InitBPlus

InitCustomBPlus

Syntax

constructor BPlusProtocol.Init(APPtr : AbstractPortPtr);

Purpose

Instantiate a B+ protocol object.

Description

This constructor initializes a BPlusProtocol object for file transfers using the default protocol options (DefProtocolOptions). It also allocates several heap buffers needed for internal use (approximately 3KB).

Example

```
var
  UP : UartPort;
  BP : BPlusProtocol;
...
  UP.InitFast(Com1, 9600);
...
  if not BP.Init(@UP) then
    {handle error}
...
  BP.Done;
```

Set up for a B+ file transfer at 9600 baud using Com1.

See Also

BPlusProtocol.InitCustom

Syntax

```
procedure InitBPlus(var P : ProtocolRecPtr; PortPtr : PortRecPtr);
```

Purpose

Allocate and initialize a protocol record.

Description

This routine initializes a protocol record for B+ file transfers using the specified port and the default protocol options (DefProtocolOptions). It also allocates several heap buffers needed for internal use (approximately 3KB).

Example

```
var
  P : PortRecPtr;
  BP : ProtocolRecPtr;
...
  InitFast(P, Com2, 9600);
...
  InitBPlus(BP, P);
  if AsyncStatus <> ecOk then
    {handle errors};
...
  DoneBPlus(BP);
```

Set up for a B+ file transfer at 9600 baud using Com2.

See Also

InitCustomBPlus

Syntax

```
constructor BPlusProtocol.InitCustom(APPtr : AbstractPortPtr; Options : Word);
```

Purpose

Instantiate a B+ protocol object with custom options.

Description

This constructor initializes a BPlusProtocol object for file transfers using the specified port and protocol Options (normally DefProtocolOptions). It also allocates several heap buffers needed for internal use (approximately 3KB).

Example

```
var
  DP : Digi14Port;
  BP : BPlusProtocol;
...
  DP.InitFast(Com5, 9600);
...
  if not BP.InitCustom(@DP, DefProtocolOptions) then
    {handle errors};
...
  BP.Done;
```

Set up for a B+ file transfer at 9600 baud using Com5.

See Also

BPlusProtocol.Init

InitCustomBPlus

Syntax

```
procedure InitCustomBPlus(var P : ProtocolRecPtr; PortPtr : PortRecPtr;  
                          Options : Word);
```

Purpose

Allocate and initialize a protocol record with custom options.

Description

This routine initializes a protocol record for B+ file transfers using the specified port and protocol Options (normally DefProtocolOptions). It also allocates several heap buffers needed for internal use (approximately 3KB).

Example

```
var  
  P : PortRecPtr;  
  BP : ProtocolRecPtr;  
...  
  InitFast(P, Com2, 9600);  
  ...  
  InitCustomBPlus(BP, P, DefProtocolOptions);  
  if AsyncStatus <> ecOk then  
    {handle errors};  
  ...  
  DoneBPlus(BP);
```

Set up for a B+ file transfer at 9600 baud using Com2.

See Also

InitBPlus

Syntax

```
{IFDEF UseStreams}
constructor BPlusProtocol.Load(var S : IdStream);
```

Purpose

Load a BPlusProtocol object from a stream.

Description

This constructor loads a BPlusProtocol object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to BPlusProtocol.Store.

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Load is called.

See the entry for XmodemProtocol.Store for a discussion of how to link a protocol object loaded from a stream with a port object.

The stream registration routine for a BPlusProtocol object is BPlusProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses
  OpRoot, ApPort, OoCom, OoAbsPcl, OoBPlus;
var
  UP : UartPort;
  BP : BPlusProtocol;
  S : BufIdStream;
  Status : Word;
begin
  if not S.Init('BPLUS.STM', SOpen, 1024) then begin
    WriteLn('Error opening stream');
    Halt;
  end;
  S.RegisterHier(BPlusProtocolStream);
  S.RegisterHier(UartPortStream);
  S.RegisterPointer(1000, @UP);
  S.Get(UP);
  S.Get(BP);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Load error: ', Status);
    Halt;
  end;
  WriteLn('Loaded successfully');
  S.Done;
  BP.Done;
  UP.Done;
end.
```

Instantiates UP and BP from a stream created by the Store example using the stream's Get method, which calls the objects' Load constructors indirectly.

Notice that a pointer to the port object is registered to ensure that BPlusProtocol.Load can create a link between the two objects.

ProcessDLE

Syntax

```
procedure ProcessDLE(P : ProtocolRecPtr; var Start, Upload : Boolean);  
procedure BPlusProtocol.ProcessDLE(var Start, Upload : Boolean);
```

Purpose

Handle <DLE> characters received while in terminal mode.

Description

Your program should call this procedure when it receives a character while in terminal mode (i.e., when not currently processing a B+ protocol). The <DLE> is usually the start of a B+ handshaking packet. ProcessDLE accepts the rest of the packet. If any errors are encountered while receiving the packet, ProcessDLE automatically requests that CompuServe retransmit that packet. Once the packet is received without errors, ProcessDLE checks the packet type and processes it accordingly. The '+' packet and the 'T' packet are acceptable; other packet types are ignored.

ProcessDLE sets Start to False after '+' packets or unexpected packets—meaning your program should *not* start the B+ protocol. ProcessDLE sets Start to True after the 'T' packet—meaning that the protocol is ready to start. Your program should then call either ProtocolTransmit or ProtocolReceive, depending on the value of Upload, to start the protocol. Or, your program could call the background protocol methods (PrepareTransmitPart and ProtocolTransmitPart or PrepareReceivePart and ProtocolReceivePart).

ProcessDLE does not return control to your program until the packet is successfully received, an error occurs, or a timeout (30 seconds) occurs.

Example

```
var  
  UP : UartPort;  
  BP : BPlusProtocol;  
  Start, Upload : Boolean;  
...  
repeat  
  if UP.CharReady then begin  
    UP.GetChar(C);  
    case C of  
      cENQ : ProcessENQ;  
      cDLE : begin  
        BP.ProcessDLE(Start, Upload);  
        if Start then begin  
          if Upload then  
            BP.ProtocolTransmit  
          else  
            BP.ProtocolReceive;  
        end;  
      end;  
    else Write(C);  
  end;  
end;  
  
(Process keyboard input)  
...  
until Finished;
```

A typical terminal loop used for starting B+ protocols. All characters are processed normally except <ENQ> and <DLE>. These characters are processed using BPlusProtocol methods. Eventually, a received <DLE> will signal the start of a protocol and ProcessDLE will set Start to True—meaning that the B+ protocol should be started.

This example shows the OOP calling syntax. The procedural syntax is ProcessDLE(P, Start, Upload) where P is a previously initialized ProtocolRecPtr.

Syntax

```
procedure ProcessENQ(P : ProtocolRecPtr);  
procedure BPlusProtocol.ProcessENQ;
```

Purpose

Handle <ENQ> characters received while in terminal mode.

Description

Your program should call this routine whenever it receives an <ENQ> character while in terminal mode (i.e., when not already processing a B+ protocol).

The <ENQ> character is a signal from CompuServe to prepare to process a B+ protocol transfer. ProcessENQ initializes several internal protocol fields and answers with the string <DLE>++<DLE>++.

Example

See the example for ProcessDLE.

See Also

ProcessDLE

ProcessESCI

Syntax

```
procedure ProcessESCI(P : ProtocolRecPtr; X, Y : Byte);  
procedure BPlusProtocol.ProcessESCI(X, Y : Byte);
```

Purpose

Handle <ESC>I sequences received while in terminal mode.

Description

If your CompuServe profile specifies VT52 terminal emulation, your program may receive the sequence <ESC>I, called the CompuServe Interrogation Sequence. Your program should return an information string telling CompuServe about your terminal capabilities. All of this information is already encoded into ProcessESCI except the dimensions of your screen (or terminal window). That information must be passed into ProcessESCI via the X and Y parameters.

Since the <ESC>I sequence consists of two characters, it must be handled a bit differently than the <ENQ> and <DLE> sequences (from your program's point of view, not the B+ protocol's). The best approach is to use the Async Professional CheckForString function as shown in the example below. Alternatively, you could use PeekCharTimeout with a very short timeout value, say 2 ticks, since the 'I' would probably follow the <ESC> within that time period.

Example

```
var  
  UP : UartPort;  
  BP : BPlusProtocol;  
  Index1 : Byte;  
...  
  Index1 := 0;  
  repeat  
    if UP.CharReady then begin  
      UP.GetChar(C);  
      case C of  
        cENQ : BP.ProcessENQ;  
        cDLE : ...  
      else begin  
        if CheckForString(Index1, C, cESC+'I') then  
          BP.ProcessESCI(80, 25)  
        else  
          Write(C);  
      end;  
    end;  
  end;  
  {Process keyboard input}  
  ...  
until Finished;
```

This example shows how to check for the <ESC>I sequence using the CheckForString function.

See Also

ProcessDLE

Syntax

```
procedure SetHandleResumeProc(P : ProtocolRecPtr; HRP : HandleResumeProc);  
procedure BPlusProtocol.SetHandleResumeProc(HRP : HandleResumeProc);
```

Purpose

Specify a custom HandleResume procedure.

Description

This routine allows you to specify a routine that will be called when ProtocolReceive detects a name collision between the incoming file and an existing file. This is your opportunity to set a specific action for this file (resume, rename, overwrite, or cancel).

If using APBPLUS, a handle resume procedure should have the form:

```
{SF+}  
procedure MyHandleResume(P : ProtocolRecPtr);  
begin  
    ...  
end;
```

P is a pointer to the ProtocolRecPtr that called this procedure. It can be used to reference internal protocol fields or it can be passed to any protocol functions that need it.

If using OOBPLUS, a handle resume procedure should have the form:

```
{SF+}  
procedure MyHandleResume(AP : AbstractProtocolPtr);  
begin  
    ...  
end;
```

See "Resuming Interrupted Transfers" earlier in this section for more information.

To remove a handle resume procedure use the built-in NoHandleResume procedure:

```
SetHandleResume(NoHandleResume);
```

When NoHandleResume is the active procedure it does *not* mean that no resumes are requested. It means that file name collisions are resolved by the current value of SetOverwriteOption.

Example

```
procedure MyHandleResume(AP : AbstractProtocolPtr);  
begin  
    with AP^ do begin  
        if Pathname = 'AUTOEXEC.BAT' then  
            SetOverwriteOption(WriteFail)  
        else  
            SetOverwriteOption(WriteResume);  
        end;  
    end;  
    ...  
    BP.SetHandleResume(MyHandleResume);
```

Sets MyHandleResume as the procedure to be called when an incoming file name collision occurs.

Store

Syntax

```
{IFDEF UseStreams}  
procedure BPlusProtocol.Store(var S : IdStream);
```

Purpose

Store a BPlusProtocol object to a stream.

Description

Store writes an image of the BPlusProtocol object to the stream S.

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The addresses of all user hooks (an AcceptFile function, for example) must be registered with the stream before Store is called.

More importantly, you must give BPlusProtocol.Store some means of forming a link between the protocol object and the port object whose address was passed to the Init constructor. See the entry for XmodemProtocol.Store for a discussion of this issue.

The stream registration routine for a BPlusProtocol object is BPlusProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses  
  OpRoot, ApPort, OoCom, OoAbsPcl, OoBPlus;  
var  
  UP : UartPort;  
  BP : BPlusProtocol;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not UP.InitFast(Com1, 2400) then begin  
    WriteLn('Error initializing port');  
    Halt;  
  end;  
  if not BP.Init(@UP) then begin  
    WriteLn('Error initializing protocol');  
    Halt;  
  end;  
  if not S.Init('BPLUS.STM', SCreate, 1024) then begin  
    WriteLn('Error initializing stream');  
    Halt;  
  end;  
  S.RegisterHier(BPlusProtocolStream);  
  S.RegisterHier(UartPortStream);  
  S.RegisterPointer(1000, @UP);  
  S.Put(UP);  
  S.Put(BP);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Store error: ', Status);  
    Halt;  
  end;  
  WriteLn('Stored successfully');  
  S.Done;  
  BP.Done;  
  UP.Done;  
end.
```


Instantiates the port and protocol objects and then stores them in a stream using the stream's Put method, which calls the object's Store method indirectly.

Notice that a pointer to the UartPort object is registered so that BPlusProtocol.Store can create a link between the two objects.

See BPlusProtocol.Load for an example program that reloads these objects.

See Also

BPlusProtocol.Load

XmodemProtocol.Store

I. Simple Text File Transfer: APASCII/OOASCII

APASCII/OOASCII provide the ability to do what is commonly called an "ASCII protocol." This is a bit of a misnomer, because in an ASCII transfer neither side of the link is following any agreed-upon rules for data transfer. An "ASCII protocol" is really just a convenient way of transmitting a text file.

A typical situation where you might use an ASCII protocol is when the machine you're communicating with doesn't support any protocols at all. For example, you might need to transfer a text file to a minicomputer that doesn't have any protocols available. One way of accomplishing this would be to attach your PC to the minicomputer as a terminal and invoke a program (on your PC) that has an ASCII protocol (like our demonstration programs SIMPCOM and OOPCOM). Then, using this program as a terminal emulator, you would navigate your way to the minicomputer's editor and open up a new text file. Then you would start an ASCII protocol transmit of the file you need to transfer. The minicomputer's editor would simply see this as keystroke input to the editor. You'd finish the transfer by saving the editor's file.

The Async Professional ASCII protocol provides options for tailoring such transfers to the remote machine's speed, such as the delays between transmitted characters and lines. For example, when transmitting a file into a remote computer's editor, you might need to insert delays to avoid overflowing the editor's keystroke buffer.

When using ASCII transfers, it is difficult for the receiver to know when the transfer is over because there is no agreed-upon method for telling the receiver that it has received all of the data. Async Professional ASCII protocol terminates on any of three conditions: when it receives a ^Z (unfortunately, the transmitter can't be counted on to send a ^Z), when it times out waiting for more data, and when the user aborts the protocol via the port abort hook you supply. All of these are considered normal terminations: the file is saved and the protocol ends. You can add other termination conditions by customizing your port abort hook.

Shared Declarations

Constants

```
apSuppressCtrlZ = $0800;
```

If this protocol option is selected and a file is being transmitted to the remote, the protocol stops sending characters when a ^Z character is found in the file (without sending the ^Z). If this option is not selected, the protocol continues sending data (including the final ^Z) until there is no more data to send. Generally, you would want to leave this option off and let the ^Z be transmitted, since the receiver might be able to recognize the ^Z as the end of the file (as Async Professional does).

```
DefBlockLen : Word = 60;
```

This is mostly a convenience for status reporting. Many of the status routines assume that the protocol is based on block transfers. Block length is also required when estimating transfer times. In ASCII protocols, the closest equivalent of a block is one line of text. DefBlockLen assigns an arbitrary value of 60 as the average line length. If this turns out not to be a good estimate, the worst that will happen is that the time estimates returned by EstimateTransferSecs (APABSPCL/OOABSPCL) and the status information on the number of blocks transferred will be inaccurate.

```
DefEOLChar : Char = cCR;
```

This is the default end-of-line character (carriage return). The ASCII protocol needs to know the proper end-of-line character so it can insert interline delays at the appropriate times. If you need to change this, modify the typed constant or call the SetEOLChar procedure.

DefInterCharDelay : Word = 0;

Default duration for delay between characters, in milliseconds. See SetDelays for more information.

DefInterLineDelay : Word = 0;

Default duration for delay between lines, in milliseconds. See SetDelays for more information.

DefRcvTimeout : Word = 364;

Default timeout period (in clock ticks) used when receiving an ASCII file. The file transfer is assumed to end when a ^Z is received or DefRcvTimeout clock ticks (364 = 20 seconds) go by without receiving any characters.

DefAsciiOptions : Word = 0;

Default ASCII protocol options.

APASCII Declarations

Types

```
AsciiProtocolPtr = ^AsciiProtocol;  
AsciiProtocol =  
  record  
    ...  
  end;
```

Record used to store data needed by the various routines in APASCII to implement the ASCII protocol.

Procedures and Functions

```
procedure PrepareReceivePartAS(P : ProtocolRecPtr);  
  {-Prepare to call the background function ProtocolReceivePartAS}  
procedure PrepareTransmitPartAS(P : ProtocolRecPtr);  
  {-Prepare to use the background function ProtocolTransmitPartAS}  
procedure ProtocolReceiveAS(P : ProtocolRecPtr);  
  {-Receive a batch of files transmitted using the ASCII protocol}  
function ProtocolReceivePartAS(P : ProtocolRecPtr) : ProtocolStateType;  
  {-Perform one increment of a background protocol receive}  
procedure ProtocolTransmitAS(P : ProtocolRecPtr);  
  {-Start ASCII protocol transmit}  
function ProtocolTransmitPartAS(P : ProtocolRecPtr) : ProtocolStateType;  
  {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the similarly named functions in APABSPCL (for example, PrepareReceivePartAS is documented in PrepareReceivePart). See §7.C for a complete discussion of these functions.

OOASCII Declarations

Types

```
AsciiProtocolPtr = ^AsciiProtocol;  
AsciiProtocol =  
  object(AbstractProtocol)  
    ...  
  end;
```

Object that defines data fields and methods needed to implement the ASCII protocol.

Procedures and Functions

```
procedure AsciiProtocol.PrepareReceivePart; virtual;  
  {-Prepare to call the background function ProtocolReceivePart}  
procedure AsciiProtocol.PrepareTransmitPart; virtual;  
  {-Prepare to use the background function ProtocolTransmitPart}  
procedure AsciiProtocol.ProtocolReceive; virtual;  
  {-Receive a batch of files transmitted using the ASCII protocol}  
function AsciiProtocol.ProtocolReceivePart : ProtocolStateType; virtual;  
  {-Perform one increment of a background protocol receive}  
procedure AsciiProtocol.ProtocolTransmit; virtual;  
  {-Start ASCII protocol transmit}  
function AsciiProtocol.ProtocolTransmitPart : ProtocolStateType; virtual;  
  {-Perform one increment of a background protocol transmit}
```

These functions are not documented here because they work exactly the same as the function with the same name in OOABSPCL. See §7.C for a complete discussion.

Syntax

destructor AsciiProtocol.Done; virtual;

Purpose

Dispose of the protocol object.

Description

This method destroys an AsciiProtocol-based object.

Example

See the example for AsciiProtocol.InitCustom.

See Also

DoneAscii

Syntax

procedure DoneAscii(var P : ProtocolRecPtr);

Purpose

Dispose of the protocol record.

Description

This routine disposes of the protocol record pointed to by P, previously allocated by a call to InitAscii or InitCustomAscii.

Example

See the examples for InitAscii and InitCustomAscii.

See Also

AsciiProtocol.Done

Syntax

function GetLineNumber(P : ProtocolRecPtr) : LongInt;
function AsciiProtocol.GetLineNumber : LongInt;

Purpose

Return the current line number.

Description

As an ASCII file transfer progresses, the protocol keeps track of how many end-of-line (EOL) characters have been received/sent. This function can be called to determine how many lines have been transferred so far. The information is primarily of use in status routines.

Example

WriteLn('Lines transferred: ', GetLineNumber(P));
Display the current line number.

See Also

SetEOLChar

SetShowStatusProc (APABSPCL)

AbstractProtocol.SetShowStatusProc

Init / InitAscii

Syntax

constructor AsciiProtocol.Init(APPtr : AbstractPortPtr);

Purpose

Allocate and initialize a protocol control block.

Description

This constructor initializes an AsciiProtocol object using the default protocol options (DefProtocolOptions). See InitCustom for further details.

Example

```
var
  UP : UartPort;
  AP : AsciiProtocol;
...
  UP.InitFast(Com1, 2400);
...
  if not AP.Init(@UP) then
    {handle error} ;
...
  AP.Done;
```

Set up for an ASCII file transfer at 2400 baud using Com1.

See Also

InitAscii

AsciiProtocol.InitCustom

Syntax

procedure InitAscii(var P : ProtocolRecPtr; PortPtr : PortRecPtr);

Purpose

Allocate and initialize a protocol control block.

Description

This routine initializes a protocol record for ASCII file transfers using the default protocol options (DefProtocolOptions). See InitCustomAscii for further details.

Example

```
var
  AP : ProtocolRecPtr;
  P : PortRecPtr;
...
  InitPortFast(P, Com1, 2400);
...
  InitAscii(AP, P);
...
  DoneAscii(AP);
```

Set up for an ASCII file transfer at 2400 baud using Com1.

See Also

AsciiProtocol.Init

InitCustomAscii

Syntax

```
constructor AsciiProtocol.InitCustom(APPtr : AbstractPortPtr;  
    InterCharDelay, InterLineDelay : Word;  
    Options : Word);
```

Purpose

Allocate and initialize a protocol control block with the specified options.

Description

This constructor initializes an AsciiProtocol object for file transfers using the specified port, delay periods (normally DefInterCharDelay and DefInterLineDelay), and protocol Options (normally DefProtocolOptions). See SetDelays for a description of the InterCharDelay and InterLineDelay parameters.

Example

```
UP.InitFast(Com1, 2400);  
...  
if not AP.InitCustom(@UP, DefInterCharDelay, DefInterLineDelay,  
    DefProtocolOptions or apSuppressCtrlZ) then  
    {handle error} ;  
...  
AP.Done;
```

Set up for an ASCII file transfer at 2400 baud using Com1. Use default delay periods and enable default protocol options, as well as the apSuppressCtrlZ option.

See Also

InitCustomAscii

SetDelays

Syntax

```
procedure InitCustomAscii(var P : ProtocolRecPtr; PortPtr : PortRecPtr;  
    InterCharDelay, InterLineDelay : Word; Options : Word);
```

Purpose

Allocate and initialize a protocol control block with the specified options.

Description

This routine initializes a protocol record for ASCII file transfers using the specified port, delay periods (normally DefInterCharDelay and DefInterLineDelay), and protocol Options (normally DefProtocolOptions). See SetDelays for a description of the InterCharDelay and InterLineDelay parameters.

Example

```
InitPortFast(P, Com1, 2400);  
...  
InitCustomAscii(AP, P, DefInterCharDelay, DefInterLineDelay,  
    DefProtocolOptions or apSuppressCtrlZ);  
...  
DoneAscii(AP);
```

Set up for an ASCII file transfer at 2400 baud using Com1. Use default delay periods and enable default protocol options, as well as the apSuppressCtrlZ option.

Load

Syntax

```
{IFDEF UseStreams}  
constructor AsciiProtocol.Load(var S : IdStream);
```

Purpose

Load an AsciiProtocol object from a stream.

Description

This constructor loads an AsciiProtocol object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to AsciiProtocol.Store.

See XmodemProtocol.Store for a discussion of how to link a protocol object loaded from a stream with a port object.

The stream registration routine for an AsciiProtocol object is AsciiProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses  
  OpRoot, ApPort, OoCom, OoAbsPcl, OoAscii;  
var  
  UP : UartPort;  
  AP : AsciiProtocol;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not S.Init('ASCII.STM', SOpen, 1024) then begin  
    WriteLn('Error opening stream');  
    Halt;  
  end;  
  S.RegisterHier(AsciiProtocolStream);  
  S.RegisterHier(UartPortStream);  
  S.RegisterPointer(1000, @UP);  
  S.Get(UP);  
  S.Get(AP);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Load error: ', Status);  
    Halt;  
  end;  
  WriteLn('Loaded successfully');  
  S.Done;  
  AP.Done;  
  UP.Done;  
end.
```

Instantiates UP and AP from a stream created by the Store example using the stream's Get method, which calls the objects' Load constructors indirectly.

Notice that a pointer to the port object is registered to ensure that AsciiProtocol.Load can create a link between the two objects.

Syntax

```
procedure SetDelays(P : ProtocolRecPtr; InterCharDelay, InterLineDelay : Word);  
procedure AsciiProtocol.SetDelays(InterCharDelay, InterLineDelay : Word);
```

Purpose

Set the delay (in milliseconds) between each character and each line.

Description

InterCharDelay is the amount of time, in milliseconds, to pause between characters when transmitting data. InterLineDelay is the amount of time, in milliseconds, to pause between lines when transmitting. Normally, both values should be set to 0 (DefInterCharDelay and DefInterLineDelay, respectively) in order to maximize performance. However, if the data is being fed directly into an application (a text editor, for example), it may be necessary to insert delays between characters or lines to allow the application time to process the data.

Example

```
SetDelays(P, 2, 50);
```

Delay two milliseconds (1/500th of a second) between characters and fifty milliseconds (1/20th of a second) between lines when transmitting data to the remote.

See Also

AsciiProtocol.InitCustom

InitCustomAscii

Syntax

```
procedure SetEOLChar(P : ProtocolRecPtr; C : Char);  
procedure AsciiProtocol.SetEOLChar(C : Char);
```

Purpose

Set the character used to mark the end of a line.

Description

This routine allows you to specify the character that marks the end of a line (the end-of-line or EOL character). The default EOL character is a carriage return (DefEOLChar = cCR = #13 = ^M). When receiving files from a Unix system, however, you would probably want to change the EOL character to a line feed (= cLF = #10 = ^J), since Unix text files use ^J to mark the end of a line, rather than the ^M^J sequence normally used for DOS text files.

Example

```
SetEOLChar(P, cLF);
```

Change the end-of-line character to a line feed.

See Also

GetLineNumber

Store

Syntax

```
{IFDEF UseStreams}  
procedure AsciiProtocol.Store(var S : IdStream);
```

Purpose

Store an AsciiProtocol object to a stream.

Description

Store writes an image of the AsciiProtocol object to the stream S.

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

You must give AsciiProtocol.Store some means of forming a link between the protocol object and the port object whose address was passed to the Init constructor. See XmodemProtocol.Store for a discussion of this issue.

The stream registration routine for an AsciiProtocol object is AsciiProtocolStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses  
  OpRoot, ApPort, OoCom, OoAbsPcl, OoAscii;  
var  
  UP : UartPort;  
  AP : AsciiProtocol;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not UP.InitFast(Com1, 2400) then begin  
    WriteLn('Error initializing port');  
    Halt;  
  end;  
  if not AP.Init(@UP) then begin  
    WriteLn('Error initializing protocol');  
    Halt;  
  end;  
  if not S.Init('ASCII.STM', SCreate, 1024) then begin  
    WriteLn('Error initializing stream');  
    Halt;  
  end;  
  S.RegisterHier(AsciiProtocolStream);  
  S.RegisterHier(UartPortStream);  
  S.RegisterPointer(1000, @UP);  
  S.Put(UP);  
  S.Put(AP);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Store error: ', Status);  
    Halt;  
  end;  
  WriteLn('Stored successfully');  
  S.Done;  
  AP.Done;  
  UP.Done;  
end.
```

Instantiates the port and protocol objects and then stores them in a stream using the stream's Put method, which calls the objects' Store methods indirectly.

Notice that a pointer to the UartPort object is registered so that AsciiProtocol.Store can create a link between the two objects.

See AsciiProtocol.Load for an example program that reloads these objects.

See Also

AsciiProtocol.Load

XmodemProtocol.Store

J. Demonstration Programs

1. File Transfer Program: FX/FXO

FX and FXO (for File Transfer) are utility programs that demonstrate virtually every protocol and protocol feature of Async Professional. While these programs are primarily for demonstration purposes they might, in fact, serve some useful purpose--such as adding an external protocol to your favorite communications program.

These are command line programs that provide one transmit or one receive protocol session. The procedural version (FX) and the object-oriented version (FXO) are used in exactly the same way. To start a Zmodem protocol transmit of all the PAS files in your directory, you would type the following:

```
fx /c 2 /b 9600 /z /t *.pas
```

FX opens Com2 at 9600 baud and transmits all PAS files using the Zmodem protocol. After all the PAS files are transmitted, FX reports the ending AsyncStatus value. If you want to transmit additional files, run FX again with the appropriate parameters.

FX expects a command line of the following form:

```
FX [options] SrcFilename
```

SrcFilename is the name of the file or file mask to be transmitted or received.

The following options can be specified (the options are not case sensitive):

/B BaudRate	Baudrate
/C #	Com port number [default = 1]
/T	Transmit mode [default]
/R	Receive mode
/S	ASCII transfer
/X	Xmodem/XmodemCRC [default]
/K	Xmodem1K
/L	Xmodem1KG
/Y	Ymodem
/G	YmodemG
/Z	Zmodem
/F	Kermit
/P	BPlus
/A	Zmodem option - resume interrupted transfer
/O	Zmodem option - always overwrite files
/N	Zmodem option - only overwrite if newer
/I	Zmodem option - skip if file doesn't exist
/D	Demonstration of background process

If you use /B to specify a baud rate, FX opens the specified port at that baud rate with no parity, 8 data bits, and 1 stop bit. If you do not specify a baud rate, FX opens the specified port without disturbing its existing parameters. You would typically omit the baud rate when you run FX using DOS EXEC from some other communications program (so that FX uses the line parameters left by the parent program).

If you specify /T to transmit files (this is the default), you must supply a file mask (or file name) as the last parameter on the command line.

Use /R to receive files. If the protocol you selected doesn't support file names (Xmodem and Ascii), then you must supply a receive file name as the last parameter. If the selected protocol does support file names, then any name you specify on the command line is ignored.

The options /S through /F specify the protocol. FX supports all standard Async Professional protocols.

/A through /I specify some Zmodem options. /A resumes interrupted file transfers. It can be specified by either the transmitter or receiver. /O tells Zmodem to always overwrite existing files (the default is to skip existing files). /N tells Zmodem to overwrite existing files only if the incoming file has a later time stamp. /I tells Zmodem not to accept files unless they already exist on the receiving machine.

The /D option allows you to use the background demonstration provided by FX. This activates a small, two-line window beneath the status window. While the protocol is in progress, anything you type at the keyboard is displayed in this window. The primary objective of the demonstration is to provide an example of how to setup and use a background process.

FX uses hardware flow control if you select a UART configuration (in APDEFINE.INC) that provides hardware flow control. FX does not use software flow control.

The progress of the protocol transfer is displayed with the "standard" protocol status display (very similar to the display used by SIMPCOM and OOPCOM). You are free to extract some or all of the source code for this display. It uses FastWrite and a few other routines of the FASTW1 unit (FASTW1 is required only if you don't have Turbo Professional or Object Professional.)

FX creates a log (FX.HIS) of all transmitted and received files. It appends to this file if it already exists. The log entry consists of the protocol type, file name, whether it was transmitted or received, and whether the transfer was successful or not. The FX logging routine also deletes partially received files unless the protocol in use is Zmodem or B+ (since they can resume the receive from the point of interruption).

If UseTracing is defined in APDEFINE.INC, FX creates a trace file, FX.TRC, of all characters received and transmitted during the protocol transfer. By default, a trace queue of 10000 characters is used.

FX uses the WriteFail option of WriteRename (specified by a call to SetOverwriteOption) for all protocols except Zmodem. When a received file name conflicts with an existing file, FX renames the incoming file by replacing the first character of the file name with '\$'. For example, FX.PAS becomes \$X.PAS.

Zmodem uses its own file management rules. FX uses the default of WriteProtect which means it skips files that already exist at the destination. You can modify this with the /O and /N command line parameters. Note that Zmodem doesn't have a rename option.

Both FX and FXO have code to demonstrate other Async Professional features (like file lists). Some of this code is currently commented out because it demonstrates conflicting or specialty features. You might want to peruse the TransferFiles procedure of each program to get an idea of what these other options look like.

Finally, note that FXO demonstrates stream I/O of protocol objects (using the Object Professional stream manager). To see this in action, define the symbol CreateStream at the top of FXO.PAS. FXO will then create and reload a stream file as it runs.

2. Background File Transfer Program: BACK/BACKO

BACK/BACKO are TSRs that demonstrate how to perform protocol transfers in a TSR. An executable file, compiled for Zmodem, is provided for BACKO. BACK is provided in source form only.

The procedural version (BACK) and the object-oriented version (BACKO) are used in exactly the same way. You choose the desired protocol for BACK/BACKO via \$DEFINE statements in the source file. To compile them, you need Object Professional or TSRs Made Easy. You can also use the TSR routines in Turbo Professional with a few easy source code changes.

To keep the size of the TSR programs reasonable, BACK/BACKO use conditional defines to control which protocols are used. The following choices are available:

```
{.$DEFINE DemoXYModem}  
{.$DEFINE DemoZmodem}  
{.$DEFINE DemoKermit}  
{.$DEFINE DemoBPlus}
```

You can activate any one of these four choices by removing the period in front of the dollar sign.

BACK/BACKO offer the following command line options:

```
/B Baudrate      300 - 115200 [Default = 2400]  
/C Comport       1 - 8 [Default = 1]  
/U              Unload TSR
```

Once installed, the following hotkeys are active:

<Ctrl><RightShift>P

Start the protocol.

<Ctrl><RightShift>T

Use the popup terminal.

<Ctrl><RightShift>S

View the protocol status.

In most cases, you'll first popup BACK's simple terminal to issue the instructions to the remote computer to start the protocol. Once the remote computer has started its transfer, press <Ctrl><RightShift>P to get BACK's main popup menu. You can do this while the terminal window is still popped up or you can remove the terminal window first. Here are the options available from the main popup menu:

<T>

Transmit a file.

<R>

Receive a file.

<A>

Abort a protocol in progress.

<K>

Kickstart the underlying program's transmitter.

Pressing <T> or <R> starts a protocol transfer at this end (first prompting for protocol options and, possibly, a file name). Pressing <A> aborts a protocol already in progress. The final option, <K>,

transmits a single character (#0) to the remote computer. This is required only if you pop up BACK over another communications program and interrupt that program while it is transmitting a buffer of data (a rare occurrence to be sure, but one that can freeze the underlying communications program).

Once the protocol is underway, you can check its progress via BACK's status popup by pressing <Ctrl><RightShift>S. To save memory, this status display isn't very fancy. It shows only the file name, bytes transferred, bytes remaining, and elapsed time. The status display remains onscreen and updates itself every two seconds until any key is pressed, at which time it is popped down and the underlying application is resumed.

Although we've put a lot of thought (and code) into making these demonstration programs as robust as possible, it is still easy to disrupt them. For example, starting any communications program while a background TSR is loaded is likely to kill the background program (since the foreground communications program will steal away the background program's serial interrupt vector). Popping up another TSR while a background file transfer is in progress can also be deadly for the same reason. The most reasonable use for background file transfers is among knowledgeable users who understand the consequences of running other programs with the TSR or in turnkey systems where only a known set of programs will be run.

Two simple terminal programs, TTERM and TTERMO, are provided as samples of how a foreground program can communicate with and manage a protocol running in a TSR. See the source code for these programs for more information.

8. Text File Device Drivers

A. Overview

APTFDD and OOTFDD are two small units that provide "text file device drivers" that use the serial port. A text file device driver is a capability of Turbo Pascal that allows your programs to take advantage of the formatting power of Read(Ln) and Write(Ln) for your I/O routines. If you're not already familiar with text file device drivers, you might want to consult your Turbo Pascal Reference Manual for more information.

The I/O routines of APTFDD and OOTFDD rely, of course, on the serial port I/O routines of APCOM and OPCOM to route all Read(Ln) and Write(Ln) requests to a serial port. This means that you can do things like:

```
R := 123.456;  
WriteLn(ComText, R:0:3);
```

Assuming ComText is a properly initialized text file, the real number R will be formatted by WriteLn and sent to the serial port as the string '123.456'. Without the text file device driver, you'd have to do something like this:

```
R := 123.456;  
Str(R:0:3, S);  
PutString(ComPort, S);
```

That is, you'd need to convert R to a string manually and transmit the converted string.

This capability is most useful when you need to transmit many such formatted strings—e.g., when sending a report to a printer or terminal connected to the serial port. Consider a more complex example where you need to format and transmit an entire line of real numbers:

```
var  
  R : array[1..4] of Real;  
...  
WriteLn(ComText, R[1]:8:4, ' ', R[2]:8:4, ' ', R[3]:8:4, ' ', R[4]:8:4);
```

In the right situations, text file device drivers can handle serial port input as well. In order for this to work, though, the input stream must follow the rules of standard Read and ReadLn for text files (see your Turbo Pascal Reference Manual for more information). To see how this might work, assume that the serial input stream looks like this:

```
123.456 1<CR>  
12.3 2<CR>  
78.0 3<CR>  
<EOF>
```

Each line consists of characters that make up a real number and an integer followed by a carriage return. Since your program probably needs to perform some arithmetic on these values, it is going to want to store them in Real and Integer variables. You could read in each value with a GetString, parse out the digits and convert them using the Val procedure. Or, you could use the ReadLn procedure of the text file device driver and let Turbo Pascal do the conversion work. Like so:

```
while (Result = 0) and not Eof(ComText) do begin  
  ReadLn(ComText, R, I);  
  Result := IOResult;  
  WriteLn(R:10:2, ' ', I);  
end;
```

Here, the ReadLn statement looks for a stream of characters that signifies a real number and an integer. The ReadLn also takes care of converting those characters into a real number (R) and an integer (I). Any errors during conversion (e.g., if the ReadLn finds any non-numeric characters) are returned in IoResult if you specify {I-} or result in run-time errors if you specify {I+}.

Using ReadLn in this fashion does have its drawbacks, however. The first problem is that it hampers your ability to handle line errors. The input routine used in the Async Professional text file device driver exits whenever it receives a line error and the Read or ReadLn returns immediately with the line error in IoResult (or a runtime error with {I+}).

When this happens, you need to take some steps to "resync" the ReadLn with the input stream. Perhaps the easiest way is to do a ReadLn(ComPort). This reads all characters up to and including the carriage return. After that, you can continue reading your data with whatever ReadLn expression you were using before (understanding, of course, that whatever data was on the previous line was lost).

The following shows how you set up and use the Async Professional text file device drivers in a non-OOP program (see EXTfDD.PAS and EXTfDDO.PAS for complete example programs):

```
var
  P : PortRecPtr;
  ComText : Text;
  R : Real;
  S : String;
  Result : Word;
begin
  InitPort(P, Com2, 4800, NoParity, 8, 1, 50, 50, DefPortOptions);
  if AsyncStatus <> ecOk then
    Abort('Failed to open port: ', AsyncStatus);

  {Assign the opened port to a text file device driver}
  AssignPortDev(ComText, P);
  Reset(ComText);

  {Read some strings from the serial port}
  repeat
    ReadLn(ComText, S);
    Result := IoResult;
    WriteLn(S);
  until (Result <> 0) or Eof(ComText);
  if Result <> 0 then
    WriteLn('ReadLn failed: ', Result);
  Close(ComText);

  {Assign the opened port to a text file device driver}
  AssignPortDev(ComText, P);
  Rewrite(ComText);

  {Write some formatted strings}
  R := 123.456;
  WriteLn(ComPort, 'This is an unformatted real number: ', R);
  WriteLn(ComPort, 'This is a formatted real number: ', R:0:3);

  Close(ComText);
  DonePort(P);
end.
```

The first step is to initialize and set up the serial port. The initialized port pointer (P) is then passed to AssignPortDev. This procedure assigns the text file ComText to use the serial port pointed to by P for all reads and writes.

After this assignment, the text file is opened for input and then the program reads in strings from the serial port until it finds a ^Z in the input stream (signaling the end of the "file").

The text file ComText is closed and reopened for output using Rewrite. Then WriteLn's formatting capabilities are used to format and output two real numbers. Finally, the text file is closed and the port pointer (P) is disposed.

The OOP format for using text file device drivers is identical in every respect except that AssignPortDev accepts a pointer to a port object instead of a pointer to a port record:

```
var
  UP : UartPort;
  ComText : Text;
...
UP.InitCustom(Com2, 4800, NoParity, 8, 1, 50, 50, DefPortOptions);
if AsyncStatus <> ecOk then
  Abort('Failed to open port: ', AsyncStatus);

{Assign the opened port to a text file device driver}
AssignPortDev(ComText, @UP);
Reset(ComPort);
...
```

After this, ReadLn and WriteLn statements to ComText work exactly as they do in the non-OOP example.

Whether you use objects here depends on the rest of your Async Professional program. There's no real advantage to having the text file device driver use an object. However, if you are already using objects in your program, then you should use OOTFDD to avoid linking in the non-OOP unit APCOM.

Shared Declarations

Constants

```
SimulateLF : Boolean = True;
```

Controls whether the input routine automatically inserts a line feed character into the input buffer whenever it sees a carriage return. Turbo Pascal's text file device driver mechanism doesn't require this line feed, but it does look for it. Given that this is serial input, you can't always guarantee that another character will be available at the port. If no characters are ready, then the device driver waits for one. This means that a call to ReadLn won't be satisfied until the next character *after* the carriage return.

Whether or not this is a problem depends on the contents of the input stream. If you know that your input stream will contain only carriage returns (rather than carriage return/line feed pairs) then you should leave SimulateLF set to True. If you know that your input stream will contain both carriage returns and line feeds, then change SimulateLF to False.

AssignPortDev

Syntax

```
procedure APTFDD.AssignPortDev(var F : Text; P : PortRecPtr);  
procedure OOTFDD.AssignPortDev(var F : Text; P : AbstractPortPtr);
```

Purpose

Assign a text file to the specified port.

Description

This procedure assigns the text file F to use the port object (or port record) pointed to by P. P *must* point to a properly opened and initialized port. The port parameter is a PortRecPtr when using APTFDD and an AbstractPortPtr when using OOTFDD.

This assignment does not open the text file. You must still call Reset to use the port in input mode, or Rewrite or Append to use the file in output mode.

All Read(Ln) and Write(Ln) operations for the text file F are then directed to the serial port specified by P. Note that you can still control the port with the normal port routines that are part of APCOM/OOCOM (e.g., to change the line parameters or flush the buffer).

If your program is compiled with {\$I-}, all line errors and type conversion errors are reported through IOResult. As with all Turbo Pascal I/O operations, failing to read IOResult after an error causes all subsequent I/O operations to be skipped.

If your program is compiled with {\$I+}, line errors and conversion errors cause a runtime error. Generally, you would always want to handle the I/O errors yourself by specifying {\$I-} and checking IOResult.

Examples

```
var  
  P : PortRecPtr;  
  ComText : Text;  
...  
  InitPortFast(P, Com1);  
  AssignPortDev(ComText, P);
```

Opens the port P to use Com1 and standard Async Professional parameters. The port pointer is then assigned to the text file ComText.

```
var  
  UP : UartPort;  
  ComText : Text;  
...  
  UP.InitFast(Com1);  
  AssignPortDev(ComText, @UP);
```

Same as the above example, but shows the format for using port objects.

Syntax

```
function APTFDD.GetPortDevPtr(var F : Text) : PortRecPtr;  
function OOTFDD.GetPortDevPtr(var F : Text) : AbstractPortPtr;
```

Purpose

Return the port pointer (object or record) associated with the specified text file.

Description

This function returns a pointer to the port record (or port object) associated with the text file F. In APTFDD this function returns a PortRecPtr. In OOTFDD this function returns an AbstractPortPtr.

Examples

```
procedure GetData(var T : Text);  
begin  
  if CheckDCD(GetPortDevPtr(T)) then  
    ReadLn(T, R1, R2, R3)  
  else  
    {handle error};  
end.
```

This routine verifies that the carrier detect (DCD) signal is present before calling ReadLn.

```
procedure GetData(var T : Text);  
begin  
  if GetPortDevPtr(T)~.CheckDCD then  
    ReadLn(T, R1, R2, R3)  
  else  
    {handle error};  
end.
```

An OOP version of the same example.

9. Terminal Emulation

A. Overview

A terminal, whether a dumb CRT attached to a host computer or a computer itself, has two basic modes of handling characters received from the host. The first is to display all characters received from the host computer. The second is to understand and respond to command sequences from the host for setting colors, setting cursor position, scrolling, and so on.

In the case of a dumb terminal attached to a host computer, the logic for understanding these command sequences (also called "escape" sequences, since <ESC> is often the first character of the sequence) is built into the terminal. When using a PC as a terminal for a host computer, the terminal program must understand these command sequences. This process is referred to as terminal emulation—since the software on the PC is emulating the behavior built into a terminal.

There are a variety of standards for these command sequences. By far the most widely used is the ANSI standard. In the PC world, IBM and Microsoft narrowed this standard by including only a subset of the complete ANSI standard in DOS's ANSI.SYS terminal emulator. Since this is the most widespread standard in the PC community, it is the one supported by Async Professional. Other standards (e.g., the complete ANSI standard and the DEC VT100 standard) are fairly similar.

The APANSI unit provides this support in a traditional procedural fashion.

The OOEMU unit (for Object-Oriented EMUulator) provides this support in an object-oriented fashion. You'll also find that OOEMU is much more flexible and more easily extended than APANSI.

B. OOP vs. Non-OOP Emulation

As with all of the add-on layers of Async Professional, terminal emulation continues the OOP and non-OOP split started back at the interface layer. Unlike other add-on layers of Async Professional, though, APANSI and OOEMU are quite different approaches to terminal emulation and they won't be documented together. Instead you'll find each discussed and documented in its own section.

Both units do, however, support the same set of ANSI escape sequences (which is the same set supported by DOS's ANSI.SYS). The table below lists the actual escape sequences and a short description of each. The general format of each sequence is an <ESC>, a left square bracket, optional numeric parameters (indicated by the "#" sign), and a command letter. Refer to your DOS User's Guide for more complete information.

ANSI Escape Sequences

#27'[#;#H'	Move cursor to row;column
#27'[#;#f'	Move cursor to row;column
#27'[f'	Move cursor to 1,1
#27'[#A'	Move cursor up # rows
#27'[#B'	Move cursor down # rows
#27'[#C'	Move cursor forward # columns
#27'[#D'	Move cursor backward # columns
#27'[s'	Save position of cursor
#27'[u'	Restore position of cursor
#27'[0J'	Clear screen below cursor
#27'[1J'	Clear screen above cursor
#27'[2J'	Clear entire screen
#27'[J'	Same as #27'[2J' (DOS) or #27'[0J' (VT100)
#27'[0K'	Clear screen from cursor to end of line
#27'[1K'	Clear screen from beginning of line to cursor
#27'[2K'	Clear current line
#27'[K'	Same as #27'[2K' (DOS) or #27'[0K' (VT100)
#27'[#;...;#m'	Set graphics rendition. # is one of the following
	0 All attributes off
	1 Bold (intensity) on
	2 Faint on (not supported)
	3 Italic on (not supported)
	4 Underscore on (treated same as 1)
	5 Blinking on
	6 Rapid blinking on (treated same as 5)
	7 Reverse video on
	8 Invisible text
	27 Reverse video off
	30 Black foreground
	31 Red foreground
	32 Green foreground
	33 Yellow foreground
	34 Blue foreground
	35 Magenta foreground
	36 Cyan foreground
	37 White foreground
	40 Black background
	41 Red background
	42 Green background
	43 Yellow background
	44 Blue background
	45 Magenta background
	46 Cyan background

47 White background
 48 Subscript background (not supported)
 49 Superscript background (not supported)
 Up to 5 parameters may be passed between the '[' and the 'm',
 with ';' used to separate them
 #27' [=#h' Set video mode to #, where # is one of the following:
 0 40x25 B&W
 1 40x25 color
 2 80x25 B&W
 3 80x25 color
 4 320x200 color (not supported)
 5 320x200 B&W (not supported)
 6 640x200 B&W (not supported)
 7 Wrap mode on (not supported--wrap always on)
 #27' [?#h' Same as #27' [=#h'
 #27' [=#l' Reset mode. Same as #27' [=#h' except parameter 7 turns wrap
 mode off rather than on
 #27' [?#l' Same as #27' [=#l'
 #27' [n6' Device Status Report. Return the current cursor position

Async Professional's ANSI emulators also support ^G (bell) and ^L (formfeed):

^G Ring bell
 ^L Clear the entire screen

C. Procedural Terminal Emulation: APANSI

APANSI provides a simple, easy-to-use ANSI emulator in a non-object oriented fashion.

Generally, in a terminal program, you would have a check for incoming characters as part of your main loop. A typical check might look something like this:

```
if CharReady then begin
  GetChar(C);
  if AsyncStatus = ecOk then
    Write(C)
  else
    {handle error}
end;
```

This displays all characters to the CRT exactly as received.

If you know that the incoming character stream contains ANSI escape sequences, this code is no longer adequate. However, you can take advantage of the APANSI emulator by replacing only one statement, the call to Write(C):

```
if CharReady then begin
  GetChar(C);
  if AsyncStatus = ecOk then
    WriteCharAnsi(C)
  else
    {handle error}
end;
```

Now, all received characters are displayed through the ANSI emulator. When the received character is a normal character, APANSI displays it in the current attribute at the current cursor location. If the received character is part of a command sequence, APANSI stores it until it receives the complete sequence. Once it has the complete command, it performs the requested action—such as changing a video attribute or moving the cursor. If it receives any errors while collecting the various command characters, or it doesn't support the requested command, it displays all of the characters in the sequence to the CRT.

For a complete example that uses WriteCharAnsi see EXANSI.PAS.

APANSI provides two simple user hooks: a ClearScreenProc and a SoundBellProc. Both of these default to built-in routines: DefClearScreen and DefSoundBell.

The ClearScreenProc is called whenever APANSI receives a ^L (ASCII FormFeed). The default routine simply calls Turbo Pascal's ClrScr procedure, which should be acceptable in most cases. However, if you need to perform a custom action whenever the screen is cleared, you should set up your own ClearScreenProc. See SetClearScreenProc for information about how to set up your own routine.

The SoundBellProc is called whenever APANSI receives a ^G (ASCII Bell). The default routine looks like this:

```
procedure DefSoundBell;
begin
  Sound(220);
  Delay(200);
  NoSound;
end;
```

This bell sound is somewhat less obnoxious than the DOS bell (the one you get when you write a ^G to the screen). If you prefer your own bell sound, or none at all, you must provide your own SoundBell routine. See SetSoundBellProc for information on how to do this.

Note that APANSI includes the Turbo Pascal CRT unit in its uses list. It automatically substitutes TPCRT or OPCRT if the appropriate define (UseTPro or UseOPro) is set in APDEFINE.INC.

ANSI Sequences Requiring a Response

Some ANSI command sequences require a response from the terminal. APANSI can recognize and respond to only one of these sequences—the Device Status Report option 6 (request cursor position). The Device Status Report request cursor position is commonly used by a host or BBS to determine if the calling terminal supports ANSI terminal emulation. When it receives the Device Status Report sequence, APANSI immediately outputs the current cursor position via the Current Position Report command. If the host receives the Current Position Report, it assumes that the terminal supports ANSI terminal emulation.

WriteCharAnsiPort and WriteStringAnsiPort allow you to specify the port for transmission of the Current Position Report response, therefore allowing use of multiple open COM ports.

If you will have only one COM port open at a time, you can choose an alternative approach. You can use SetCurrentAnsiPort to specify the port and then use the WriteCharAnsi and WriteStringAnsi procedures. These procedures do not require you to specify the port as a parameter, but instead use the current port set by SetCurrentAnsiPort.

Declarations

Constants

CurrentAnsiPort : PortRecPtr = nil;

Points to the port that APANSI uses when it needs to transmit. This is set by SetCurrentAnsiPort.

InhibitAutoLF : Boolean = True;

If InhibitAutoLF is False, APANSI generates automatic line feeds. This means that when a character is written to column 80, APANSI moves the cursor to column 1 of the next line. Since some hosts issue their own <CR><LF>, this could create the effect of moving down two lines rather than one. Therefore the default is InhibitAutoLF = True so that APANSI will not generate the automatic line feed.

InhibitModeChange : Boolean = True;

Prevents the remote from changing the local video mode. If you want to honor remote ANSI requests to change the video mode (e.g. from color to monochrome), change this to False.

MaskBlinkOnInverse : Boolean = True;

MaskBlinkOnInverse, when True, masks the blink bit after a request to invert an attribute. When False, a request to invert an attribute results in a blinking attribute if the previous foreground attribute was using an intense color and the monitor is using blink mode.

UseVT100Mode : Boolean = False;

Specifies how the defaults for the ANSI sequences ED and EL ('J' and 'K') are determined. When UseVT100Mode is True, the default for missing parameters is zero. When UseVT100Mode is False, the default for missing parameters is 2.

Types

ClearScreenProc = procedure;

A procedure called to clear the screen when a ^L (ASCII FF) character is received.

SoundBellProc = procedure;

A procedure called to "ring the bell" when a ^G (ASCII BEL) character is received.

SetClearScreenProc

Syntax

```
procedure SetClearScreenProc(CSP : ClearScreenProc);
```

Purpose

Set a ClearScreen procedure to be called for formfeed characters.

Description

This routine allows you to specify a procedure that is to be called to clear the screen when a ^L character (ASCII FF) is received from the remote. The default routine (not interfaced by APANSI) is as follows:

```
procedure DefClearScreen;  
begin  
  ClrScr;  
end;
```

Example

```
{ $F+ }  
procedure MyClearScreen;  
var SaveCheckSnow : Boolean;  
begin  
  SaveCheckSnow := CheckSnow;  
  CheckSnow := False;  
  ClrScr;  
  CheckSnow := SaveCheckSnow;  
end;  
...  
SetClearScreenProc(MyClearScreen);
```

An alternative ClearScreen procedure that ensures that snow checking is turned off when the screen is cleared (assumes use of TPCRT or OPCRT).

Syntax

```
procedure SetCurrentAnsiPort(P : PortRecPtr);
```

Purpose

Set the current ANSI port for any required ANSI output.

Description

This procedure sets the default output port used by any ANSI sequences that require a response. (Currently, the only supported sequence is the Device Status Report, which requests the terminal to report its current cursor position.)

This is the most convenient method of specifying an ANSI output port when your program has only one open port. After setting the current port, all calls to `WriteCharAnsi` and `WriteStringAnsi` will use that port, if necessary, when responding to ANSI requests.

If you do not call `SetCurrentAnsiPort`, `WriteCharAnsi` and `WriteStringAnsi` will not respond to any ANSI requests that require an answer (such requests are simply ignored).

This procedure can be used even with the OOP interface layer. The call would look like this:

```
var
  ComPort : UartPort;
...
ComPort.Init(...);
SetCurrentAnsiPort(ComPort.PR);
```

Example

```
var
  PR : PortRecPtr;
...
  SetCurrentAnsiPort(PR);
```

Specifies PR as the current ANSI port.

See Also

`WriteCharAnsi`
`WriteCharAnsiPort`

`WriteStringAnsi`
`WriteStringAnsiPort`

Syntax

```
procedure SetSoundBellProc(SBP : SoundBellProc);
```

Purpose

Set a `SoundBell` procedure to be called on Bell characters.

Description

This routine specifies a procedure that is called to "ring the bell" when a `^G` character (ASCII BEL) is received. The default routine (not interfaced by APANSI) is:

```
procedure DefSoundBell;
begin
  Sound(220); Delay(200); NoSound;
end;
```

Example

```
{ $F+ }
procedure MySetSoundBellProc;
begin
  Write(^G);
end;
...
SetSoundBellProc(MySetSoundBellProc);
```

An alternative `SoundBell` procedure that uses CRT/TPCRT/OPCRT's standard facility for ringing the bell.

WriteCharAnsi / WriteCharAnsiPort

Syntax

```
procedure WriteCharAnsi(C : Char);
```

Purpose

Write a character to the screen and handle ANSI escape sequences.

Description

Normally, this routine simply writes the character C to the screen. If the character initiates or continues an ANSI escape sequence, however, the character is processed in the same manner as the ANSI.SYS device driver would process it.

WriteCharAnsi can handle an ANSI sequence that requires a response (such as Device Status Report). However, it can support only one open port. This is accomplished by first calling SetCurrentAnsiPort to specify the port. A better alternative to SetCurrentAnsiPort/WriteCharAnsi is to use WriteCharAnsiPort, which can handle multiple open ports.

A complete list of ANSI escape sequences is at the beginning of §9.B.

Examples

```
WriteCharAnsi('^L');
```

Clear the screen.

```
WriteCharAnsi('^G');
```

Ring the bell.

See Also

SetCurrentAnsiPort
WriteCharAnsiPort

WriteStringAnsi

Syntax

```
procedure WriteCharAnsiPort(P : PortRecPtr; C : Char);
```

Purpose

Write a character to the screen and handle ANSI escape sequences. Supports multiple ANSI ports.

Description

This function writes character C to the screen. If an incoming ANSI sequence requires a response, the port P is used to transmit that response.

This function is generally preferred over SetCurrentAnsiPort/WriteCharAnsi because it can handle multiple open ports.

This procedure can be used even with the OOP interface layer. The call would look something like this:

```
var
  ComPort : UartPort;
...
ComPort.Init(...);
ComPort.GetChar(C);
WriteCharAnsiPort(ComPort.PR, C);
```

Example

```
var
  PR : PortRecPtr;
...
WriteCharAnsiPort(PR, $07);
```

Ring the bell using the port PR.

See Also

WriteStringAnsiPort

Syntax

```
procedure WriteStringAnsi(S : String);
```

Purpose

Write a string to the screen and handle ANSI escape sequences.

Description

This routine simply calls `WriteCharAnsi` repeatedly, passing it each character in string `S`. `WriteStringAnsi` can handle an ANSI sequence that requires a response (such as Device Status Report). However, it can support only one open port. This is accomplished by first calling `SetCurrentAnsiPort` to specify the port. A better alternative to `SetCurrentAnsiPort/WriteStringAnsi` is to use `WriteStringAnsiPort`, which can handle multiple open ports.

Examples

```
WriteStringAnsi('^L'Hello world!');
```

Clears the screen and writes 'Hello world!' at row 1, column 1.

```
const
  BgnInverseVideo : string[4] = #27'[7m';
  EndInverseVideo : string[5] = #27'[27m';
...
  WriteStringAnsi(BgnInverseVideo);
  WriteStringAnsi(...);
  WriteStringAnsi(EndInverseVideo);
```

Write a string in inverse video.

See Also

`WriteCharAnsi`

`WriteStringAnsiPort`

Syntax

```
procedure WriteStringAnsiPort(P : PortRecPtr; S : String);
```

Purpose

Write a string to the screen and handle ANSI escape sequences. Supports multiple ANSI ports.

Description

This procedure writes a string to the screen. If the string contains an ANSI sequence that requires a response, the port `P` is used to transmit that response. This procedure is generally preferred over `SetCurrentAnsiPort/WriteStringAnsi` because it can handle multiple open ports.

This procedure can be used even with the OOP interface layer, like this:

```
var
  ComPort : UartPort;
...
  ComPort.Init(...);
  ComPort.GetString(S, ...);
  WriteStringAnsiPort(ComPort.PR, S);
```

Examples

```
...
var
  PR : PortRecPtr;
...
  WriteStringAnsiPort(PR, BgnInverseVideo);
  WriteStringAnsiPort(...);
  WriteStringAnsiPort(PR, EndInverseVideo);
```

Write a string in inverse video using port `PR`.

D. Object-Oriented Terminal Emulation: OOEMU

OOEMU implements a terminal emulator using an object-oriented design. Its underlying philosophy is somewhat different than APANSI's. OOEMU *never* writes to the screen. Instead, it evaluates the current character and tells you what it is or tells you to ignore the character if it is part of an incoming command sequence. This behavior is preferred in those cases where you are using a window manager (such as Object Professional or Turbo Vision) to actually display the characters.

Here's the object hierarchy provided by OOEMU:



The TerminalEmulator defines the basic data structures and commands used by an emulator. It doesn't do any translation of incoming characters, however, so you generally wouldn't instantiate such an object.

The AnsiEmulator adds the logic for processing ANSI escape sequences. When you pass characters to the AnsiEmulator, it tells you whether they are normal characters that you should display, whether they are part of an incoming sequence that you should ignore, or whether a complete command has been received and you should perform an action such as moving the cursor.

Typically, your programs would instantiate and use an AnsiEmulator. If you want to provide another type of emulation, derive your own emulator from TerminalEmulator and add the appropriate logic for processing the incoming command strings.

Here's a nearly complete program that shows how you might use the AnsiEmulator:

```
program ExEmu; {EXEMU.PAS}
{-A simple ANSI terminal}
uses
  Crt, ApMisc, ApPort, ApUart, OoCom, OoEmu;
var
  UP : UartPort;
  AE : AnsiEmulator;
  CR : CommandRecord;
  C : Char;
  Finished : Boolean;
begin
  {Open a port}
  UP.InitCustom(Com2, 2400, NoParity, 8, 1, 500, 500, DefPortOptions);
  if AsyncStatus <> ecOk then begin
    WriteLn('Failed to open port: ', AsyncStatus);
    Halt;
  end;

  {Instantiate an emulator}
  AE.Init(32);

  {Simple terminal -- quit on <AltX>}
  Finished := False;
  repeat
    {Process chars to send}
    if KeyPressed then begin
      C := ReadKey;
      if C = #0 then
        Finished := ReadKey = #$2D
      else if UP.TransReady then
```



```

        UP.PutChar(C);
    end;

    {Process chars received}
    if UP.CharReady then begin
        UP.GetChar(C);
        if AsyncStatus <> ecOk then begin
            WriteLn('M`J'Line error ', AsyncStatus);
            UP.FlushInBuffer;
        end else begin
            AE.ProcessChar(C, CR);
            case CR.Cmd of
                eNone      : {do nothing} ;
                eChar       : Write(CR.Ch);
                eClearScreen : ClrScr;
                eGotoXY      : GotoXY(CR.X, CR.Y);
                {... handle other commands ...}
            end;
        end;
    end;
until Finished;
UP.Done;
AE.Done;
end.

```

This program is built along the same lines as other terminal programs—it instantiates the required objects and drops into a loop for processing incoming and outgoing characters.

Every received character is passed to the emulator via the call to `AE.ProcessChar`, which returns a command record (CR). The `Cmd` field of the command record tells you what to do next. If the received character is a normal character that you should display, `Cmd` is `eChar`. If the received character is part of an ANSI sequence, `Cmd` is `eNone`—meaning you should just ignore this character for now. If the character is the last one in an ANSI command sequence, then `Cmd` tells you what that command is: `eGotoXY`, `eClearScreen`, and so on (see the Constants section for a complete list of possible commands).

If the command sequence is a Device Status Request, `Cmd` is `eDeviceStatusReport`. You must format and output a Current Position Report in response.

As you can see, OOEMU puts a lot more of the burden on your program than APANSI did. However, this is precisely what makes OOEMU more flexible. For example, if you're constraining your terminal I/O to just one part of the screen, you don't want to blindly follow an ANSI request to position the cursor at row one, column one. Your program should adjust those coordinates to make them relative to your window's location on the screen. Similarly, if your terminal window is currently obscured by another window, you don't want to automatically display all characters that you receive (because you'd just overwrite what was covering your terminal window). These and a host of other issues make OOEMU the best choice when you are using a windowing library like Object Professional. `TerminalWindow` (discussed in the next section) handles all of these issues for you and provides a terminal window that fits in with the Object Professional window hierarchy.

Declarations

Constants

<code>eNone</code>	<code>= 0;</code>	<code>{no command, ignore this char}</code>
<code>eChar</code>	<code>= 1;</code>	<code>{no command, process the char}</code>
<code>eGotoXY</code>	<code>= 2;</code>	<code>{absolute goto cursor position call}</code>
<code>eUp</code>	<code>= 3;</code>	<code>{cursor up}</code>

```

eDown          = 4; {cursor down}
eRight         = 5; {cursor right}
eLeft         = 6; {cursor left}
eClearBelow    = 7; {clear screen below cursor}
eClearAbove    = 8; {clear screen above cursor}
eClearScreen   = 9; {clear entire screen}
eClearEndOfLine = 10; {clear from cursor to end of line}
eClearStartOfLine = 11; {clear from cursor to the start of line}
eClearLine     = 12; {clear entire line that cursor is on}
eSetMode       = 13; {set video mode}
eSetBackground = 14; {set background attribute}
eSetForeground = 15; {set foreground attribute}
eSetAttribute  = 16; {set video attribute (foreground and background)}
eSaveCursorPos = 17; {save cursor position}
eRestoreCursorPos = 18; {restore cursor position}
eDeviceStatusReport = 19; {report device status or cursor position}
eError        = 255; {parser error}

```

Emulator commands.

```

Escape      = #27;
LeftBracket = #91;
Semicolon   = #59;
FormFeed    = #12;

```

Special parser characters for ANSI escape sequences.

```

teMapVT100 = $0001;
teMaskBlinkOnInverse = $0002;

```

Emulator options. If the teMapVT100 option is set, the AnsiEmulator object assumes a parameter of 0 (as VT100 terminals do) rather than 2 (as ANSI.SYS does) when the J or K commands are issued with no parameters. (See the chart of ANSI escape sequences in §9.B.) To ensure proper behavior of the emulator, this option should always be set when emulating a VT100.

If the teMaskBlinkOnInverse option is set, the blink bit is masked after a request to invert an attribute. If it is not set, a request to invert an attribute results in a blinking attribute if the previous foreground attribute was using an intense color and the monitor is using blink mode.

Types

```

AnsiEmulatorPtr = ^AnsiEmulator;
AnsiEmulator =
  object(TerminalEmulator)
  ...
end;

```

Emulator object for PC ANSI codes.

```

TerminalEmulatorPtr = ^TerminalEmulator;
TerminalEmulator =
  object(Root)
  ...
end;

```

Basic terminal emulator object. Makes no changes to incoming characters. Defines data fields and methods shared in common by "real" terminal objects.

```

CommandRecord =
  record
    Ch : Char; {the character}
    Cmd : Byte; {the command}
    X, Y, Z : Byte; {the parameters (only X and Y used currently)}
  end;

```

Record type used to describe a command and its parameters. See the entry for TerminalEmulator.ProcessChar.

Syntax

```
destructor TerminalEmulator.Done; virtual;
```

Purpose

Dispose of a TerminalEmulator.

Description

This destructor disposes of the command queue allocated by TerminalEmulator.Init.

Example

See the example for TerminalEmulator.Init.

Syntax

```
constructor TerminalEmulator.Init(QueueSize : Byte);
```

Purpose

Initialize a TerminalEmulator.

Description

This constructor initializes the TerminalEmulator's internal data fields and allocates a queue large enough to hold QueueSize characters. This queue is used to temporarily store any characters received by ProcessChar as part of an escape sequence. When the escape sequence is complete, the emulator translates the characters in the queue into a command code and empties the queue.

When using an object that doesn't override ProcessChar, a QueueSize of 1 is sufficient, since TerminalEmulator.ProcessChar itself always returns a command code of eChar. For an AnsiEmulator object, a QueueSize of 32 is generally sufficient.

Example

```
if not TE.Init(32) then  
  {handle error} ;  
...  
TE.Done;
```

Initialize a TerminalEmulator with a queue size of 32 and later dispose of it.

See Also

TerminalEmulator.Done

AnsiEmulator.Init

Syntax

```
{IFDEF UseStreams}  
constructor TerminalEmulator.Load(var S : IdStream);
```

Purpose

Load a TerminalEmulator object from a stream.

Description

This constructor loads an TerminalEmulator object from a stream. This would not normally be called. The only time you would need to call it is when you derive a new object from TerminalEmulator (you would call TerminalEmulator.Load from your derived Load constructor).

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to TerminalEmulator.Store.

The stream registration routine for a TerminalEmulator object is TerminalEmulatorStream.

See the Object Professional documentation for more information about object-oriented streams.

See Also

TerminalEmulator.Store

Syntax

```
procedure TerminalEmulator.ProcessChar(C : Char;  
                                       var Command : CommandRecord); virtual;
```

Purpose

Process a character and return a CommandRecord.

Description

This routine, which is typically overridden by descendants of TerminalEmulator, is called each time a character is received from the remote in order to translate it into an emulator command. C is the character received, and Command is a record variable containing information about the command.

If the character is not part of an escape sequence, Command.Cmd is set to eChar on return, and the character to be displayed on the screen is in Command.Ch (normally the same as C, but not necessarily).

If the character begins or continues an escape sequence, Command.Cmd is set to eNone on return, and the character should be ignored.

If the character ends an escape sequence, Command.Cmd contains the code for the command it represents, and the parameters passed as part of the command (if any) are in Command.X, Command.Y, and/or Command.Z. If the parameter is a single value that is not an X or Y screen coordinate, the value is stored in Command.X.

If the emulator does not recognize the escape sequence, Command.Cmd is set to eError.

Example

```
var  
    TE : TerminalEmulator;  
    CR : CommandRecord;  
...  
    TE.ProcessChar('A', CR);
```

On return, CR.Ch is 'A' and CR.Cmd is eChar.

See Also

TerminalEmulator.Init

AnsiEmulator.ProcessChar

Syntax

```
{ $IFDEF UseStreams }  
procedure TerminalEmulator.Store(var S : IdStream);
```

Purpose

Store an TerminalEmulator object to a stream.

Description

Store writes an image of the TerminalEmulator object to the stream S. This would not normally be called. The only time you would need to call it is when you derive a new object from TerminalEmulator (you would call TerminalEmulator.Store from your derived Store method).

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The stream registration routine for an TerminalEmulator object is TerminalEmulatorStream.

See the Object Professional documentation for more information about object-oriented streams.

See Also

TerminalEmulator.Load

teOptionsAreOn / teOptionsOff / teOptionsOn

Syntax

```
function TerminalEmulator.teOptionsAreOn(Options : Word) : Boolean;
```

Purpose

Return True if all specified options are on.

Description

This function returns True if the specified emulator option(s) are currently selected.

Example

```
if teOptionsAreOn(teMapVT100) then
  teOptionsOff(teMapVT100)
else
  teOptionsOn(teMapVT100);
Toggle the state of the teMapVT100 option.
```

Syntax

```
procedure TerminalEmulator.teOptionsOff(Options : Word);
```

Purpose

Turn options off.

Description

This procedure deactivates the specified emulator option(s).

Example

See the example for teOptionsAreOn.

Syntax

```
procedure TerminalEmulator.teOptionsOn(Options : Word);
```

Purpose

Turn options on.

Description

This procedure activates the specified emulator option(s).

Example

See the example for teOptionsAreOn.

Done / Init

Syntax

destructor `AnsiEmulator.Done; virtual;`

Purpose

Dispose of an `AnsiEmulator`.

Description

This destructor disposes of the command queue allocated by `AnsiEmulator.Init`.

Example

See the example for `AnsiEmulator.Init`.

See Also

`AnsiEmulator.Init`

Syntax

constructor `AnsiEmulator.Init(QueueSize : Byte);`

Purpose

Initialize an `AnsiEmulator`.

Description

This constructor initializes the `AnsiEmulator`'s internal data fields and allocates a queue large enough to hold `QueueSize` characters. A `QueueSize` of 32 is generally sufficient.

Example

```
if not AE.Init(32) then  
  {handle error} ;
```

```
...
```

```
AE.Done;
```

Initialize an `AnsiEmulator` with a queue size of 32 and later dispose of it.

See Also

`TerminalEmulator.Init`

Syntax

```
{$IFDEF UseStreams}  
constructor AnsiEmulator.Load(var S : IdStream);
```

Purpose

Load an AnsiEmulator object from a stream.

Description

This constructor loads an AnsiEmulator object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to AnsiEmulator.Store.

The stream registration routine for an AnsiEmulator object is AnsiEmulatorStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses  
  OpRoot, OoEmu;  
var  
  AE : AnsiEmulator;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not S.Init('ANSIEMU.STM', SOpen, 1024) then begin  
    WriteLn('Error opening stream');  
    Halt;  
  end;  
  S.RegisterHier(AnsiEmulatorStream);  
  S.Get(AE);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Load error: ', Status);  
    Halt;  
  end;  
  WriteLn('Loaded successfully');  
  S.Done;  
  AE.Done;  
end.
```

Instantiates AE from a stream created by the Store example using the stream's Get method, which calls the object's Load constructor indirectly.

See Also

AnsiEmulator.Store

ProcessChar

Syntax

```
procedure AnsiEmulator.ProcessChar(C : Char;  
                                   var Command : CommandRecord); virtual;
```

Purpose

Process a character and return a CommandRecord.

Description

This routine is called each time a character is received from the remote in order to translate it into an emulator command. C is the character received, and Command is a record variable containing information about the command.

If the character is not part of an escape sequence, Command.Cmd is set to eChar on return, and the character to be displayed on the screen is in Command.Ch.

If the character begins or continues an escape sequence, Command.Cmd is set to eNone on return, and the character should be ignored.

If the character ends an escape sequence, Command.Cmd contains the code for the command it represents, and the parameters passed as part of the command (if any) are in Command.X, Command.Y, and/or Command.Z. If the parameter is a single value that is not an X or Y screen coordinate, the value is stored in Command.X.

If the emulator does not recognize the escape sequence, Command.Cmd is set to eError.

Example

```
var  
  AE : TerminalEmulator;  
  CR : CommandRecord;  
  S : string;  
  I : Byte;  
...  
S := ^L'Hello world!';  
for I := 1 to Length(S) do begin  
  AE.ProcessChar(S[I], CR);  
  case CR.Cmd of  
    eClearScreen : ClrScr;  
    eChar : Write(CR.Ch);  
    {... handle other commands ...}  
  end;  
end;
```

When I = 1, CR.Cmd is eClearScreen. Thereafter, CR.Cmd is eChar and CR.Ch contains the I'th character in S.

See Also

TerminalEmulator.ProcessChar

Syntax

```
{IFDEF UseStreams}
procedure AnsiEmulator.Store(var S : IdStream);
```

Purpose

Store an AnsiEmulator object to a stream.

Description

Store writes an image of the AnsiEmulator object to the stream S.

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The stream registration routine for an AnsiEmulator object is AnsiEmulatorStream.

See the Object Professional documentation for more information about object-oriented streams.

Example

```
uses
  OpRoot, OoEmu;
var
  AE : AnsiEmulator;
  S : BufIdStream;
  Status : Word;
begin
  if not AE.Init(32) then begin
    WriteLn('Error initializing emulator');
    Halt;
  end;
  if not S.Init('ANSIEMU.STM', SCreate, 1024) then begin
    WriteLn('Error initializing stream');
    Halt;
  end;
  S.RegisterHier(AnsiEmulatorStream);
  S.Put(AE);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error: ', Status);
    Halt;
  end;
  WriteLn('Stored successfully');
  S.Done;
  AE.Done;
end.
```

Instantiates the emulator object and then stores it in a stream using the stream's Put method, which calls the object's Store method indirectly.

See AnsiEmulator.Load for an example program that reloads this object.

See Also

AnsiEmulator.Load

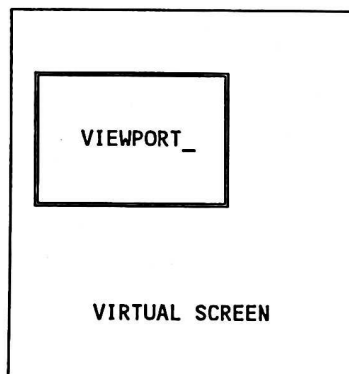
E. Terminal Window Object: TERMWIN

The TERMWIN unit combines the lower-level facilities in Async Professional with those in Object Professional to create a pair of objects that can display incoming and outgoing text within a window that can be scrolled, moved, zoomed, or resized. The TerminalWindow object, derived from the Object Professional CommandWindow, provides this basic functionality, and the CaptureTerminalWindow object adds the ability to capture all incoming characters to a file. The object hierarchy looks like this:



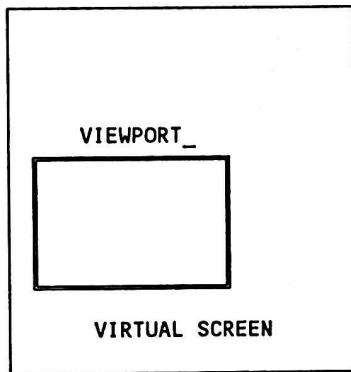
TerminalWindow's display is scrollable because it draws all text to a virtual screen (also known as a "scroll back buffer") and then copies it as appropriate to the physical screen. This virtual screen is typically as wide as the physical screen and several times higher (the default height is 200 lines).

When you issue a cursor movement command within a TerminalWindow, the display is scrolled simply by changing the region of the virtual screen that is displayed within the window on the physical screen. This region of the virtual screen is known as the "viewport," and its location is expressed in terms of its top left corner. For example, if the viewport is set to 3,7 (X,Y), then the top left corner of the window on the physical screen will display the character on row 7, column 3 of the virtual screen. (The height and width of the viewport are determined by the dimensions of the window.) The following diagram illustrates the relationship between the viewport and the virtual screen:



Keep in mind that data may continue to be received from the remote after the display is scrolled. Normally the next character received is written to the location on the virtual screen that corresponds to the position of the cursor on the physical screen. But what happens when the display has been scrolled (i.e., the viewport has been moved) far enough that this location is no longer visible on the physical screen? Where is the text written? And where is the physical cursor displayed?

Answer: The incoming text is written to the virtual screen right where it would have been if the display hadn't been scrolled, at the position of the "virtual cursor." Consider the diagram above, and notice in particular the underscore ('_') following "VIEWPORT," which represents both the location of the physical cursor within the window on screen and the location of the virtual cursor within the virtual screen. Now suppose that the display is scrolled downward such that the top of the viewport is below the virtual cursor:



Although the next incoming character is written to the virtual screen at the position of the virtual cursor, that character is not visible on the physical screen until 1) the display is scrolled upward such that the virtual cursor falls within the viewport again or 2) incoming text moves the virtual cursor back within the viewport. And what of the physical cursor? It is turned off automatically as long as the virtual cursor is outside the viewport. When the virtual cursor reenters the viewport, the physical cursor reappears.

Up to this point, it was assumed that the viewport can be changed only by the user, but that isn't always the case. By default, an option called "floating view" is enabled. When this option is in effect, the viewport is moved automatically to try to keep incoming text visible on the physical screen whenever possible. This option is especially important when the viewport is much smaller than the size assumed by the incoming data.

The floating view option is actually two options, `twFloatingHView` and `twFloatingVView`, only the second of which is enabled by default. `twFloatingVView` automatically scrolls the display vertically when the virtual cursor starts to leave the viewport, while `twFloatingHView` scrolls it horizontally. Although both options are needed to ensure that the cursor is always visible, in most cases the overall effect of `twFloatingHView` is annoying: when long lines of text are coming in quickly, the display will be scrolled almost constantly, especially if the viewport is relatively small. In general, it is best to enable only the `twFloatingVView` option, so that the display is scrolled only when the cursor moves to a new line.

You may be wondering at this point if floating view doesn't cause a problem, given that both the user and incoming text may change the viewport. And it does, if the user tries to scroll the display while text is still coming in. Although there are several possible solutions to the problem, the recommended one is: provide the user a hotkey that can be pressed to examine the contents of the scroll back buffer while text is still coming in. When this hotkey is first pressed, use the Async Professional flow control facilities to send the remote a request to stop sending data. The user can then adjust the viewport without a fight. When the hotkey is pressed a second time, send the remote a message telling it to resume its transmission. (Note that if you are not using flow control in your program, you can achieve much the same effect with a hotkey that toggles floating view on and off.)

As its name implies, a `TerminalWindow` may be associated with a `TerminalEmulator`, allowing it (for example) to respond to ANSI escape sequences for positioning the cursor, changing colors, etc. This possibility raises a problem, however, because it means that the remote can send the terminal window a command telling it to "clear the screen." Should the entire virtual screen be cleared when this happens, or just a portion of it? If the latter, how big should that region be?

TerminalWindow addresses this problem in two ways. First, it provides an option called "advance on clear" (twAdvanceOnClear). If this option is off, a clear screen command clears the entire virtual screen, resets the viewport to 1,1 and moves the virtual cursor to 1,1. If it is on, the command moves both the virtual cursor and the viewport to the top left corner of the next "page" and then clears the page, leaving previously received text intact. By default, each page is the same size as the physical screen, but you can change the page size in cases where, for example, the display is in 40x25 mode and the remote is known to assume an 80x25 screen.

A Simple Example

Now that you have a pretty good idea of what a terminal window is and how it works, the following example shows how to incorporate one into a program. This program dials a phone number specified on the DOS command line and then displays incoming text within a full-screen terminal window.

```

program TWTEST; {TWTEST.PAS}
uses
  OpRoot, ApMisc, ApPort, OoCom, OoModem, OoEmu,
  OpCrt, OpCmd, OpWindow, TermWin;
var
  UP : UartPort;
  HM : HayesModem;
  TW : TerminalWindow;
  AE : AnsiEmulator;
  Finished : Boolean;
begin
  {check for number to dial}
  if ParamCount = 0 then begin
    WriteLn('Usage: TWTEST nnn-nnnn');
    Halt;
  end;

  {open port}
  if not UP.InitFast(Com1, 1200) then begin
    WriteLn('Failed to initialize port: ', AsyncStatus);
    Halt(1);
  end;

  {instantiate a modem object}
  if not HM.Init(@UP) then begin
    WriteLn('Failed to initialize modem: ', AsyncStatus);
    Halt(1);
  end;

  {initialize terminal window}
  if not TW.Init(1, 1, ScreenWidth, ScreenHeight, @UP) then begin
    WriteLn('Failed to initialize terminal window: ', InitStatus);
    Halt(1);
  end;

  {allow <AltX> to be used to exit from program, as well as <AltF3>}
  TerminalCommands.AddCommand(ccQuit, 1, $2D00, 0);

  {initialize emulator and attach to terminal window}
  if not AE.Init(32) then begin
    WriteLn('Failed to initialize terminal emulator: ', AsyncStatus);
    Halt(1);
  end;
  TW.SetTerminalWinEmulator(@AE);

```

```

{dial the number}
Write('Dialing ', ParamStr(1), '...');
HM.DialModem( ParamStr(1) );
if AsyncStatus <> ecConnect then begin
    WriteLn('M^J'Call could not be established');
    Halt;
end;

{get rid of the "Dialing..." message}
Write('M');
ClrEol;

{process commands}
Finished := False;
repeat
    TW.Process;
    case TW.GetLastCommand of
        ccQuit, ccError : Finished := True;
    end;
until Finished;

{hang up modem}
HM.HangupModem(mOnHook, False);

{destroy objects}
TW.Done;
HM.Done;
AE.Done;
UP.Done;
end.

```

The first few lines of the program simply check to make sure that the user entered a phone number on the command line. If not, the program aborts with a help message.

The next two sections attempt to open a port and instantiate a modem object. Obviously, a real application program couldn't assume that the modem was a 1200 baud modem attached to Com1, but for these purposes the assumption is acceptable. (Just change the call to InitFast as necessary if you want to try out the program.) Note that the TerminalWindow knows nothing about the modem object, which is needed here only to dial the phone number and to hang up the modem when the call is finished.

The next step is to instantiate the terminal window. The Init constructor takes only five parameters. The first four specify the coordinates for the window (and hence the dimensions of the viewport), and the last is the address of the port to be used (giving the terminal window the means to send and receive characters). The call to TerminalCommands.AddCommand is completely optional. It simply allows <AltX> to be used to exit from the program, in addition to <AltF3>, which is the default keystroke to close a TerminalWindow. Note that, unlike most other CommandWindow-based objects, TerminalWindow does not (by default) allow <Esc> to be used to close the window.

The next step is to initialize an AnsiEmulator, which is needed if the terminal window must respond to ANSI escape sequences. Once the object is instantiated successfully, "attach" it to the terminal window by calling the SetTerminalWinEmulator method. If no terminal emulator is in use, TerminalWindow interprets all incoming text as ordinary characters, with the exceptions of ^H (backspace), ^M (carriage return), and ^J (line feed).

The next section of code tries to dial the phone number specified by the user. If successful, the "Dialing..." message is removed from the screen before the terminal window is displayed. If not, an error message is displayed and the program aborts.

Finally, the example activates the terminal window. The main loop of the program calls the `Process` method repeatedly until an exit command of `ccQuit` or `ccError` is returned. In this simple example, these are the only exit commands that are even possible, so the loop never executes more than once. But in a real application, you would likely be defining your own exit commands (`ccUser0`, etc.), in which case you would need to place your call to `Process` within a loop.

When the loop is finished, the modem is hung up. Although this is unnecessary if the session with the remote ended normally, it terminates the session cleanly in cases where the user aborted the program by pressing `<AltX>` or `<AltF3>`.

The final step destroys all objects. Although these calls are not strictly required in this particular case, it is wise to get in the habit of always destroying all the objects you create, if only to avoid forgetting to do so in cases where it is necessary.

One final note. Before attempting to compile this example program, or any other program that uses `TERMWIN`, be sure that you have `'UseOPRO'` defined in your `APDEFINE.INC` file. If it is not defined, you get a compilation error when you try to compile the `TERMWIN` unit.

Capturing a Terminal Window Session

Suppose that in the example program you had wanted to be able to keep a record of an online session. How would you do it? The answer is surprisingly simple. Make the `TW` variable an object of type `CaptureTerminalWindow`, rather than `TerminalWindow`, and add the line

```
TW.Capture(True);
```

just before the main loop. All the characters received from the remote would then be stored in the default capture file, `CAPTURE.LOG`. (Note that a slightly modified version of `TWTEST` that does this is provided under the name `CTWTEST.PAS`.)

If you want to store the data in a different file, use the `InitCustom` constructor, which allows you to specify the name of the capture file when the object is instantiated, or call the `SetCaptureFileName` method to change the name later. `InitCustom` also lets you specify the size of the capture buffer, which defaults to 128 characters (the minimum size). This buffer is used to store outgoing characters temporarily before committing them to disk. When the buffer becomes full or capture is turned off, it is automatically flushed to disk.

The state of the `ctwOverwrite` option determines what happens if the capture file already exists when the `Capture` method is called. If it is off, as it is by default, the existing capture file is reopened, and the new data is appended to the end of it. If it is on, the file is overwritten.

Although you might think that any text editor or file browser could be used to view the contents of a capture file, this is not always true. If the capture file contains ANSI control sequences, the file is difficult to read with an ordinary editor or browser. You can, however, use one of the supplied sample programs to make sense of its contents. The `TERMTEST /F` option provides a convenient means of viewing such files. To view the default capture file, for example, enter

```
TERMTEST /Fcapture.log
```

at the DOS command line. For more information on `TERMTEST` see §9.F.

Note that `CaptureTerminalWindow` *does not* capture any of the characters sent to the remote, only those received from it. If you wish to add this capability to your program, you can derive a new object from `CaptureTerminalWindow` and override the `SendOutgoingChar` method. Your method would be implemented as follows:

```

procedure MyCaptureTerminalWindow.SendOutgoingChar(C : Char);
begin
  CaptureTerminalWindow.SendOutgoingChar(C);
  if CaptureIsOn then
    ctwPutCapture(C);
end;

```

For further discussion of the `SendOutgoingChar` method, see "User Hooks," later in this section. `ctwPutCapture` is an internal (undocumented) method of `CaptureTerminalWindow` that simply puts the specified character in the capture buffer. This is probably the only situation in which you would ever need to call it.

Other Special Features

A few other special features of the `TerminalWindow` are worth mentioning here. The most notable is the full-screen zoom option (`twFullScreenZoom`). If this option is on, as it is by default, a terminal window automatically sheds its border when it is zoomed, allowing it to occupy the entire screen (or as much of it as its position and size limits allow). When the window is unzoomed, its border automatically returns.

Another useful feature is the `twWrapInWin` option. If this option is on, text is wrapped automatically when it reaches the edge of the window. For example, if the window (viewport) is 50 characters wide, the cursor automatically advances to the beginning of the next line when the cursor goes beyond the 50th column. Be forewarned that this option should generally not be used when the window is zoomable/resizeable or when ANSI escape sequences are being received.

Finally, `TerminalWindow` provides a set of four options designed to help you deal with the differing conventions concerning carriage returns and line feeds. If the `twOutCRtoCRLF` option is on, any outgoing CR (^M) characters are converted to CRLF (^M^J) pairs before they are sent to the remote. If the `twInCRtoCRLF` option is on, any CR (^M) characters received from the remote are converted to CRLF pairs. If the `twOutLFtoCRLF` option is on, any outgoing LF (^J) characters are converted to CRLF pairs. And if the `twInLFtoCRLF` option is on, any LF (^J) characters received are converted to CRLF pairs. Although these options generally aren't necessary when communicating with a BBS or information service, it is wise to provide the user some means of activating them when needed.

User Hooks

`TerminalWindow` provides two hooks that allow you to extend the object's functionality in ways that would otherwise be difficult. The first is the virtual method `GetIncomingChar`, which you can override if you want to get control as soon as a character is received from the remote. `CaptureTerminalWindow`, for example, overrides this method in order to write received characters to the capture buffer. The second user hook is the virtual method `SendOutgoingChar`, which you can override if you want to get control each time a character is sent to the remote. (An example of such a case is given above, in "Capturing a Terminal Window Session.") Both of these methods are discussed more fully later in this section.

One of the standard hooks available in all `CommandWindow`-based objects, the background task hook, is effectively disabled by `TerminalWindow`. `TerminalCommands.SetGetKeyProc` cannot be used to install a routine that performs background tasks (such as updating an on-screen clock) while waiting for keyboard input. You must use the `TerminalCommands.SetKeyPressedProc` hook instead, because `TerminalWindow` performs its own background task (checking for incoming characters) while waiting for keyboard input. Keep in mind that your background task must execute quickly, and that you should perform it only once per call to your `KeyPressed` function. If it takes too long, the `TerminalWindow` will not be able to respond promptly to incoming characters.

The Importance of Flow Control

When using a TerminalWindow to display incoming text, it is almost always necessary to enable flow control (either hardware or software). If flow control is not in use and the remote sends data continuously as fast as it can, and if TerminalWindow cannot keep up, characters are lost. Even if you have tested your application thoroughly without enabling flow control and convinced yourself that it isn't necessary, keep in mind that problems might still arise if the program is run on a slower machine or if the baud rate is increased. For best results, always use flow control with TerminalWindows.

TerminalWindow Commands

The commands available while within a terminal window are arranged by category in the list below. The first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command.

ccLeft <Left>

Scroll the display left one column.

ccRight <Right>

Scroll the display right one column.

ccUp <Up>

Scroll the display up one row.

ccDown <Down>

Scroll the display down one row.

ccHome <Home>

Scroll to the left edge of the virtual screen.

ccEnd <End>

Scroll to the right edge of the virtual screen.

ccPageUp <PgUp>

Scroll the display up one page (= height of window).

ccPageDn <PgDn>

Scroll the display down one page (= height of window).

ccTopOfFile <^PgUp>

Scroll to the beginning of the virtual screen.

ccEndOfFile <^PgDn>

Scroll to the end of the virtual screen.

ccMouseSel <ClickLeft>

Does nothing inside terminal window itself. Used to operate scroll bars, select hot spots, etc.

ccHelp <F1>, <ClickBoth>

If a user-written help routine was established by calling TerminalCommands.SetHelpProc, pressing <F1> calls that routine; otherwise this command does nothing. If there is a GetHelpProc, the terminal window passes it the following three parameters: ucTermWin, @Self, and the window's help index (specified with SetHelpIndex).

ccQuit <AltF3>, <ClickRight>

Close the terminal window.

Declarations

Constants

BadTerminalWinOptions : LongInt =

twWasBordered+twCursorIsOn+twMouseCursor+twZoomed+ctwCaptureOn;

TerminalWindow options that exist for internal use, and cannot be altered by calling twOptionsOn or twOptionsOff.

ccIncomingChar = ccUser54;

A special user command used internally.

DefCaptureBufferSize : Word = MinCaptureBufferSize;

Default size of the capture buffer used by CaptureTerminalWindow.

DefCaptureFileName = 'CAPTURE.LOG';

Default name of the capture file used by CaptureTerminalWindow.

DefScrollBackCols : Word = 80;

Default number of columns for a TerminalWindow's virtual screen.

DefScrollBackRows : Word = 200;

Default number of scroll back rows for a TerminalWindow.

DefTerminalWinOptions : LongInt =

twFloatingVView+twAdvanceOnClear+twFullScreenZoom;

The default options for a TerminalWindow.

MinCaptureBufferSize = 128;

Minimum size of the capture buffer for a CaptureTerminalWindow.

TerminalKeyMax = 106;

TerminalKeyID : string[18] = 'terminal key array';

TerminalKeySet : array[0..TerminalKeyMax] of Byte = (...);

TerminalCfgEnd : Byte = 0;

TerminalKeyId is an ID string used to mark the beginning of the configuration data area for TERMWIN; TerminalCfgEnd marks the end of the configuration data area. Between them is TerminalKeySet, the command table used by TerminalCommands. TerminalKeyMax is the last valid index into the table.

twFloatingHView = \$00000001;	{horiz view follows cursor through vscreen}
twFloatingVView = \$00000002;	{vertical view follows cursor through vscreen}
twAdvanceOnClear = \$00000004;	{advance view automatically on ClrScr}
twShowOutgoing = \$00000008;	{echo outgoing characters}
twWrapInWin = \$00000010;	{wrap text within window}
twOutCRtoCRLF = \$00000020;	{convert outgoing ^M to ^M^J}
twInCRtoCRLF = \$00000040;	{convert incoming ^M to ^M^J}
twOutLFtoCRLF = \$00000080;	{convert outgoing ^J to ^M^J}
twInLFtoCRLF = \$00000100;	{convert incoming ^J to ^M^J}
twMapMono = \$00000200;	{map ANSI/VT100 attributes to mono}
tw7Bits = \$00000400;	{strip high bit of incoming chars}
twFullScreenZoom = \$00000800;	{eliminate border when zooming}
ctwOverwrite = \$00001000;	{capture should overwrite existing file}
ctwCaptureOn = \$08000000;	{internal flags}
twWasBordered = \$10000000;	
twCursorIsOn = \$20000000;	
twMouseCursor = \$40000000;	
twZoomed = \$80000000;	

The available options for TerminalWindows. If the twFloatingHView option is on, the viewport is changed automatically when characters are received to ensure that the column containing the virtual cursor is visible on screen. If the twFloatingVView option is on, the viewport is changed automatically to ensure that the row containing the virtual cursor is visible on screen. Note that

the `twFloatingHView` option by itself is undesirable. It is best to choose only the `twFloatingVView` option, or both options, or neither option.

The `twAdvanceOnClear` option determines what happens to the virtual screen when a `Clear Screen` command is received from the remote (e.g., a `^L`). If it is off, the entire virtual screen is cleared, the viewport is reset to 1,1, and the virtual cursor is moved to 1,1. If it is on, the virtual cursor and the viewport are moved to the top left corner of the next "page" and the page is cleared, leaving previously received text intact.

If the `twShowOutgoing` option is on, any characters entered in the terminal window by the user are echoed to the screen before they are sent to the remote.

If the `twWrapInWin` option is on, displayed text is wrapped as necessary to make it all visible within the window (e.g., if the window is 50 columns wide, when the 51st character on a line comes in, it is displayed at the beginning of the next line). If the option is off, text is wrapped (if necessary) when it reaches the right edge of the virtual screen (typically the same width as the physical screen), and the window must be scrolled horizontally to view text beyond the right edge of the window.

If the `twOutCRtoCRLF` option is on, any outgoing CR (`^M`) characters are converted to CRLF (`^M^J`) pairs before they are sent to the remote. If the `twInCRtoCRLF` option is on, any CR (`^M`) characters received from the remote are converted to CRLF pairs.

If the `twOutLFtoCRLF` option is on, any outgoing LF (`^J`) characters are converted to CRLF (`^M^J`) pairs before they are sent to the remote. (This option is generally useful only when using a `TerminalWindow` descendant to send characters from files created on a UNIX system.) If the `twInLFtoCRLF` option is on, any LF (`^J`) characters received from the remote are converted to CRLF (`^M^J`) pairs. (This option is generally useful only when communicating with a UNIX system.)

The `twMapMono` option is useful if an `AnsiEmulator` is in use and a program is running on a monochrome display. If this option is on and a color change command is received from the remote, the `TerminalWindow` uses `OPCRT`'s color mapping routines to alter the requested color for best results on the monochrome display.

If the `tw7Bits` option is on, the high bit of all incoming characters is automatically stripped.

If the `twFullScreenZoom` option is on, the border for the terminal window disappears when it is zoomed, and the frame coordinates are adjusted automatically to allow the zoomed window to occupy the entire screen.

The `ctwOverwrite` option applies only to `CaptureTerminalWindows`. It determines what happens if `Capture` is asked to open a capture file that already exists. If the `ctwOverwrite` option is On, the file is overwritten. If not, new capture data is simply appended to the existing file.

The remaining options are intended strictly for internal use by `TERMWIN`.

```
ucTermWin = 40;
```

Unit code passed to the help routine by `TerminalWindow.ProcessSelf`.

Types

```
CaptureTerminalWindowPtr = ^CaptureTerminalWindow;  
CaptureTerminalWindow =  
    object(TerminalWindow)  
        ...  
    end;
```

A `TerminalWindow` with the ability to capture incoming characters and store them in a file.

```
TerminalWindowPtr = ^TerminalWindow;  
TerminalWindow =  
  object(CommandWindow)  
  ...  
end;
```

Terminal window object derived from an Object Professional CommandWindow. It displays incoming text in a moveable, zoomable, resizable window.

Variables

```
{IFDEF UseDrag}  
TerminalCommands : DragProcessor;  
{ELSE}  
TerminalCommands : CommandProcessor;  
{ENDIF}
```

The default CommandProcessor for a TerminalWindow.

CopyVirtToFile / Done

Syntax

```
procedure TerminalWindow.CopyVirtToFile(StartLine, EndLine : Integer;  
                                         FName : String;  
                                         StripTrailing : Boolean;  
                                         AsciiText : Boolean;  
                                         AppendIfExists : Boolean);
```

Purpose

Copy the specified lines of the virtual screen to a file.

Description

This routine copies a portion of the TerminalWindow's virtual screen to a file (FName). StartLine and EndLine specify the lines to be copied to the file. If AsciiText is True, only the characters on each line are written to the file, and a CRLF is appended to each line before it is written. If AsciiText is False, both video attributes and characters are written. If StripTrailing and AsciiText are both True, trailing blanks are stripped from each line before the CRLF is appended. If the specified file already exists and AppendIfExists is True, the new data is appended to it; otherwise the existing file is overwritten.

Errors are reported to the TerminalWindow's error handler in the standard Object Professional fashion. Besides the I/O errors that might occur when creating the file, the only possible error is ecBadCoordinates, which indicates that either StartLine or EndLine is invalid (e.g., EndLine is beyond the end of the scrollbar buffer, or StartLine > EndLine).

Example

```
TW.Init(1, 1, ScreenWidth, ScreenHeight, @MyUartPort);
```

```
...
```

```
TW.CopyVirtToFile(1, DefScrollBackRows, 'TERMWIN.DMP', True, True, False);
```

Dumps all the text in the virtual screen to an ASCII text file, TERMWIN.DMP, which is overwritten if it already exists.

Syntax

```
destructor TerminalWindow.Done; virtual;
```

Purpose

Dispose of a TerminalWindow.

Description

This destructor disposes of the virtual screen used to buffer scrolled data, then calls the parent's destructor (CommandWindow.Done) to dispose of screen buffers, etc.

Example

See the example for InitCustom.

See Also

InitCustom

GetIncomingChar / GetTerminalWinEmulator

Syntax

```
function TerminalWindow.GetIncomingChar(var Key : Char) : Boolean; virtual;
```

Purpose

Return True if an incoming character has arrived at the serial port.

Description

This function is called by ProcessSelf to get the next character received from the remote. If there is one, GetIncomingChar returns True and the character is in Key. If there isn't, it returns False.

GetIncomingChar is reserved for internal use: it should not be called directly. You may, however, wish to override it. CaptureTerminalWindow, for example, overrides GetIncomingChar so that it can write the received character to the capture buffer. TERMTEST's NewTerminalWindow overrides GetIncomingChar in order to simulate the receipt of characters from the remote by returning characters read out of a file. OOPCOM's OopComTerminalWindow overrides GetIncomingChar in order to return characters that were stored in a buffer while the window is inactive or busy doing something else.

Example

See the NewTerminalWindow.GetIncomingChar method in TERMTEST.PAS.

See Also

ProcessSelf

SendOutgoingChar

Syntax

```
function TerminalWindow.GetTerminalWinEmulator : TerminalEmulatorPtr;
```

Purpose

Return a pointer to an emulator object.

Description

This function returns a pointer to the terminal emulator object being used by the terminal window, if any. If no emulator is in use, GetTerminalWinEmulator returns nil.

Examples

```
TW.Init(1, 1, ScreenWidth, ScreenHeight, @MyUartPort);
```

```
...
```

```
TEP := GetTerminalWinEmulator;
```

Returns nil in TEP.

```
TW.SetTerminalWinEmulator(@MyEmulator);
```

```
...
```

```
TEP := GetTerminalWinEmulator;
```

Returns @MyEmulator in TEP.

See Also

Init

InitCustom

SetTerminalWinEmulator

GetTerminalWinPort / Init

Syntax

function TerminalWindow.GetTerminalWinPort : AbstractPortPtr;

Purpose

Return a pointer to the port object.

Description

This function returns a pointer to the Port object used by the TerminalWindow.

Example

```
TW.Init(1, 1, ScreenWidth, ScreenHeight, @MyUartPort);  
...  
APP := TW.GetTerminalWinPort;  
Returns @MyUartPort in APP.
```

See Also

InitCustom

SetTerminalWinPort

Syntax

constructor TerminalWindow.Init(X1, Y1, X2, Y2 : Byte; Port : AbstractPortPtr);

Purpose

Create a TerminalWindow.

Description

This constructor instantiates a TerminalWindow using the default window options (DefWindowOptions), the default color set (DefaultColorSet), no terminal emulator (nil), and the default terminal window options (DefTerminalWinOptions). The scroll back buffer will be DefScrollBackRows high and DefScrollBackCols wide. See InitCustom for further details.

Example

```
var  
  UP : UartPort;  
  TW : TerminalWindow;  
...  
  UP.InitFast(Com1, 2400);  
...  
  if not TW.Init(1, 1, ScreenWidth, ScreenHeight, @UP) then  
    {handle error} ;  
...  
  TW.Done;
```

Initialize a full-screen terminal window, communicating with the remote via Com1 at 2400 baud.

See Also

InitCustom

SetTerminalWinEmulator

Syntax

```

constructor TerminalWindow.InitCustom(X1, Y1, X2, Y2 : Byte;
                                     var Colors : ColorSet;
                                     WindowOptions : LongInt;
                                     Port : AbstractPortPtr;
                                     Emu : TerminalEmulatorPtr;
                                     ScrollBackRows : Word;
                                     ScrollBackCols : Word;
                                     TerminalOptions : LongInt);

```

Purpose

Create a TerminalWindow with custom options.

Description

This constructor instantiates a TerminalWindow using the specified WindowOptions and Colors. X1, Y1, X2, and Y2 are the coordinates of the interior portion of the window.

Port is a pointer to the port object to be used to communicate with the remote. Emu is a pointer to the TerminalEmulator object to be used to process incoming characters. If no terminal emulation is desired, Emu can be nil. TerminalOptions is a bit-mapped LongInt that specifies the desired terminal window options (see the Declarations section).

ScrollBackRows and ScrollBackCols determine the height and width of the virtual screen to be used as a scroll back buffer. The ScrollBackRows (height) should be at least as large as the height of the window, and is generally larger. ScrollBackCols is typically equal to ScreenWidth, although it may be larger if the display is in 40-column mode or the remote is assuming a wider display (e.g., 132 columns). ScrollBackRows * ScrollBackCols can't exceed 32760 (65521 div 2). If it does, InitCustom fails with InitStatus (not AsyncStatus) set to ecScrollBackTooBig. If there is insufficient memory to allocate the virtual screen, it fails with the error ecOutOfMemory. Unless there is a problem in allocating the virtual screen, InitCustom fails only if the parent's constructor (CommandWindow.InitCustom) fails.

Only the TextColor and TextMono fields of the specified ColorSet have special meaning for a TerminalWindow. These fields determine the default color for the virtual screen (and hence the interior of the window). That color can change, however, if a terminal emulator is in use, since the remote can send a command that resets the text color.

Example

```

var
  UP : UartPort;
  AE : AnsiEmulator;
  TW : TerminalWindow;
...
UP.InitFast(Com1, 2400);
AE.Init(32);
...
if not TW.InitCustom(
  1, 1, ScreenWidth, ScreenHeight, {window coordinates}
  DefaultColorSet,                 {colors}
  DefWindowOptions,                 {window options}
  @UP,                             {port object}
  @AE,                             {terminal emulator}
  DefScrollBackRows,                {200 rows of scroll back}
  DefScrollBackCols,               { and 80 columns}
  DefTerminalWinOptions) then      {terminal window options}
  (handle error) ;
...
TW.Done;

```

Initialize a full-screen terminal window, communicating with the remote via Com1 at 2400 baud, with support for ANSI control sequences.

Load

Syntax

```
{IFDEF UseStreams}  
constructor TerminalWindow.Load(var S : IdStream);
```

Purpose

Load a TerminalWindow object from a stream.

Description

This constructor loads a TerminalWindow object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to TerminalWindow.Store.

The addresses of both the terminal emulator object (if any) and the port object must be registered with the stream before Load is called.

The stream registration routine for a TerminalWindow object is TerminalWindowStream.

Example

```
uses  
  OpRoot, ApPort, OoCom, OpCrt, OpCmd, OpFrame, OpWindow, TermWin;  
var  
  UP : UartPort;  
  TW : TerminalWindow;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not S.Init('TERMWIN.STM', SOpen, 1024) then begin  
    WriteLn('Error opening stream');  
    Halt;  
  end;  
  S.RegisterHier(TerminalWindowStream);  
  S.RegisterHier(UartPortStream);  
  S.RegisterPointer(1000, @UP);  
  S.Get(UP);  
  S.Get(TW);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Load error: ', Status);  
    Halt;  
  end;  
  S.Done;  
  
  TW.Process;  
  
  TW.Done;  
  UP.Done;  
end.
```

Instantiates UP and TW from a stream created by the Store example using the stream's Get method, which calls Load indirectly. As in the Store example, RegisterPointer is called to register the address of the port object prior to loading the terminal window, so that the link between the two can be reestablished.

Syntax

procedure TerminalWindow.ProcessSelf; virtual;

Purpose

Begin processing a TerminalWindow.

Description

TerminalWindow's ProcessSelf method follows the general model described in the Object Professional manual under CommandWindow.Process with the following additions and exceptions:

- It handles the commands described in "Terminal Window Commands" at the start of this section.

- The cursor movement commands (<Up>, <Down>, etc.) do not affect the position of the "virtual cursor" (its position is affected only by incoming and outgoing characters). Rather, they change the viewport: the region of the virtual screen that is currently visible within the window.

- ProcessSelf exits on any line errors or any conditions set in a port abort function. The last command (returned by CommandWindow.GetLastCommand) is set to ccError and AsyncStatus is set to the condition that caused the exit.

Example

```
TW.Init(1, 1, 80, 25, @MyUartPort);  
...  
TW.Process;  
TW.Done;
```

Instantiate a TerminalWindow, process incoming and outgoing characters, then erase the window and destroy the object.

Syntax

procedure TerminalWindow.SendOutgoingChar(C : Char); virtual;

Purpose

Send a character out the serial port.

Description

This procedure is called when a character needs to be sent to the remote. It is intended strictly for internal use: it should not be called directly. You may, however, wish to override it. TERMTEST's NewTerminalWindow, for example, overrides it in order to prevent the output buffer from overflowing when testing with no device attached, and to count the total number of characters "sent" to the remote.

Example

See the NewTerminalWindow.SendOutgoingChar method in TERMTEST.PAS.

See Also

GetIncomingChar

SetPageSize / SetTerminalWinEmulator

Syntax

```
procedure TerminalWindow.SetPageSize(PageHeight : Integer);
```

Purpose

Set the height of a page in the virtual screen.

Description

This routine allows you to specify the height of the region that represents the terminal's physical screen from the remote's point of view. The default page size is equal to the height of the physical screen (usually 25, 43, or 50 lines). In some cases, however, the remote might be sending out text a "page" at a time and pausing at the end of the page (CompuServe does this, for example), and it might assume a page size of 23 or 24 lines. SetPageSize allows you to change the terminal window's idea of what constitutes a page so that it is in agreement with the remote. This is especially important if the remote is sending ANSI escape sequences, since the page size setting determines how far the terminal window needs to scroll the display when an ANSI clear screen instruction is received and the twAdvanceOnClear option is on.

Example

```
TW.SetPageSize(ScreenHeight-2);
```

Set the page size to the height of the physical screen, less the two lines used at the top and bottom for a window border.

Syntax

```
procedure TerminalWindow.SetTerminalWinEmulator(NewEmu : TerminalEmulatorPtr);
```

Purpose

Change the internal emulator object.

Description

This routine tells a terminal window object to use a different terminal emulator object. It is especially important to call this method if the emulator object originally passed to the terminal window's constructor was deallocated and then reallocated, since you cannot assume that the object will be allocated at the same place on the heap the second time.

Example

```
var
  AE : AnsiEmulator;
  TW : TerminalWindow;
...
  TW.Init(1, 1, 80, 25, @MyUartPort);
  AE.Init(32);
  TW.SetTerminalWinEmulator(@AE);
```

Set TW to use the AnsiEmulator AE.

See Also

GetTerminalWinEmulator
Init

InitCustom

Syntax

```
procedure TerminalWindow.SetTerminalWinPort(NewPort : AbstractPortPtr);
```

Purpose

Change the internal port object.

Description

This routine tells a terminal window object to use a different port object. It is especially important to call this method if the port object originally passed to the terminal window's constructor was deallocated and then reallocated, since you cannot assume that the object will be allocated at the same place on the heap the second time.

Example

```
UPP1 := New(UartPortPtr, InitFast(Com1, 2400));
UPP2 := New(UartPortPtr, InitFast(Com2, 2400));
TW.Init(1, 1, 80, 25, UPP1);
...
TW.SetTerminalWinPort(UPP2);
```

Initialize TW for use with Com1; later switch to Com2.

See Also

GetTerminalWinPort

InitCustom

Syntax

```
procedure TerminalWindow.SetView(X, Y : Integer);
```

Purpose

Set the viewport into the virtual screen.

Description

This routine resets the terminal window's viewport. Y is the row of the virtual screen to be displayed at the top edge of the window, and X is the column of the virtual screen to be displayed at the left edge.

Example

```
SetView(1, 1);
Set the viewport to 1,1.
```

Store

Syntax

```
($IFDEF UseStreams)
procedure TerminalWindow.Store(var S : IdStream);
```

Purpose

Store a TerminalWindow object to a stream.

Description

Store writes an image of the TerminalWindow object to the stream S.

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The addresses of both the terminal emulator object in use (if any) and the port object in use must be registered with the stream before Store is called.

The stream registration routine for a TerminalWindow object is TerminalWindowStream.

Example

```
uses OpRoot, ApPort, OoCom, OpCrt, OpCmd, OpFrame, OpWindow, TermWin;
var
  UP : UartPort;
  TW : TerminalWindow;
  S : BufIdStream;
  Status : Word;
begin
  if not UP.InitFast(Com1, 2400) then begin
    WriteLn('Error initializing port');
    Halt;
  end;
  if not TW.Init(2, 2, ScreenWidth-1, ScreenHeight-1, @UP) then begin
    WriteLn('Error initializing terminal window');
    Halt;
  end;
  TW.WOptionsOn(WBordered);
  if not S.Init('TERMWIN.STM', SCreate, 1024) then begin
    WriteLn('Error initializing stream');
    Halt;
  end;
  S.RegisterHier(TerminalWindowStream);
  S.RegisterHier(UartPortStream);
  S.RegisterPointer(1000, @UP);
  S.Put(UP);
  S.Put(TW);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error: ', Status);
    Halt;
  end;
  S.Done;

  TW.Process;

  TW.Done;
  UP.Done;
end.
```

Instantiates UP and TW, and stores both objects in a stream using the stream's Put method, which calls Store indirectly. RegisterPointer is called to register the address of the port object prior to storing the terminal window, so that the link between the two can be reestablished when the objects are reloaded. See Load for an example program that reloads these objects.

Syntax

```
function TerminalWindow.twOptionsAreOn(Options : LongInt) : Boolean;
```

Purpose

Return True if all specified options are on.

Description

This function returns True if the specified terminal window option(s) are currently selected.

Example

```
if twOptionsAreOn(twFloatingVView) then
  twOptionsOff(twFloatingVView)
else
  twOptionsOn(twFloatingVView);
```

Toggle the state of the twFloatingVView option.

Syntax

```
procedure TerminalWindow.twOptionsOff(Options : LongInt);
```

Purpose

Deactivate multiple options.

Description

This procedure deactivates the specified terminal window option(s), excluding those designated as BadTerminalWinOptions.

Example

See the example for twOptionsAreOn.

Syntax

```
procedure TerminalWindow.twOptionsOn(Options : LongInt);
```

Purpose

Activate multiple options.

Description

This procedure activates the specified terminal window option(s), excluding those designated as BadTerminalWinOptions.

Example

See the example for twOptionsAreOn.

WriteChar / WriteString

Syntax

procedure TerminalWindow.WriteChar(C : Char); virtual;

Purpose

Write a character to the TerminalWindow with emulation.

Description

This routine writes a single character to the TerminalWindow at the current position of the virtual cursor. If a terminal emulator is in use, the character is passed to the emulator for possible translation before it is written to the screen.

Examples

```
TW.WriteChar(`L);
```

Clears the screen if an AnsiEmulator is in use.

```
TW.WriteChar('A');
```

Writes an 'A' at the current position of the virtual cursor.

See Also

InitCustom

SetTerminalWinEmulator

WriteString

Syntax

procedure TerminalWindow.WriteString(S : String);

Purpose

Write a string to the TerminalWindow with emulation.

Description

This routine writes the specified string to the TerminalWindow at the current position of the virtual cursor. If a terminal emulator is in use, the characters are passed to the emulator for possible translation before they are written to the screen.

Example

```
TW.WriteString(`L'Hello world!'`M`J);
```

Clears the screen (if an AnsiEmulator is in use), writes 'Hello world!' at the current position of the virtual cursor and moves the cursor to the beginning of the next line.

See Also

WriteChar

Syntax

```
procedure CaptureTerminalWindow.Capture(On : Boolean);
```

Purpose

Turn data capture on or off.

Description

This routine turns capture of received data on or off. If On is True and capture isn't already activated, the capture file is opened. If capture was already on, the data in the capture buffer is flushed to disk. If On is False, the data in the capture buffer is flushed to disk and the capture file is closed.

The `ctwOverwrite` option determines what happens if Capture is asked to open a capture file that already exists. If the `ctwOverwrite` option is On, the file is overwritten. If not, new capture data is simply appended to the existing file.

Example

```
CTW.Capture(True);
```

```
...
```

```
CTW.Capture(False);
```

Turn capture on and, later, off.

See Also

[InitCustom](#)

[SetCaptureFileName](#)

Syntax

```
function CaptureTerminalWindow.CaptureIsOn : Boolean;
```

Purpose

Return True if data capture is on.

Description

This function determines whether or not the capture option has been activated successfully.

Example

```
CTW.Capture(True);
```

```
...
```

```
IsOn := CTW.CaptureIsOn;
```

CaptureIsOn returns True.

Syntax

```
destructor CaptureTerminalWindow.Done; virtual;
```

Purpose

Dispose of a CaptureTerminalWindow.

Description

This destructor disposes of the capture buffer, then calls the parent's destructor (`TerminalWindow.Done`) to dispose of the virtual screen and destroy the window. If the capture file is open when it is called, the capture buffer is flushed and the capture file closed.

GetCaptureFileName / Init

Syntax

```
function CaptureTerminalWindow.GetCaptureFileName : String;
```

Purpose

Get the name of the capture file.

Description

This function returns the name of the file used to store captured data. Note that GetCaptureFileName does not give any indication of whether or not the capture file is actually open; to obtain that information, you must call CaptureIsOn.

Example

```
CTW.Init(1, 1, ScreenWidth, ScreenHeight, @MyUartPort);  
...  
FName := CTW.GetCaptureFileName;  
This returns 'CAPTURE.LOG' (DefCaptureFileName).
```

See Also

CaptureIsOn
InitCustom

SetCaptureFileName

Syntax

```
constructor CaptureTerminalWindow.Init(X1, Y1, X2, Y2 : Byte;  
Port : AbstractPortPtr);
```

Purpose

Create a CaptureTerminalWindow.

Description

This constructor instantiates a CaptureTerminalWindow using the default window options (DefWindowOptions), the default color set (DefaultColorSet), no terminal emulator (nil), and the default terminal window options (DefTerminalWinOptions). The scroll back buffer is DefScrollBackRows high and DefScrollBackCols wide. The capture buffer will hold DefCaptureBufferSize (128) characters. See InitCustom for further details.

Example

```
var  
  UP : UartPort;  
  CTW : TerminalWindow;  
...  
  UP.InitFast(Com1, 2400);  
...  
  if not CTW.Init(1, 1, ScreenWidth, ScreenHeight, @UP) then  
    {handle error} ;  
...  
  CTW.Done;
```

Initialize a full-screen terminal window, communicating with the remote via Com1 at 2400 baud. The capture buffer will hold 128 characters at a time.

See Also

InitCustom

TerminalWindow.SetTerminalWinEmulator

Syntax

```

constructor CaptureTerminalWindow.InitCustom(X1, Y1, X2, Y2 : Byte;
                                             var Colors : ColorSet;
                                             WindowOptions : LongInt;
                                             Port : AbstractPortPtr;
                                             Emu : TerminalEmulatorPtr;
                                             ScrollBackRows : Word;
                                             ScrollBackCols : Word;
                                             TerminalOptions : LongInt;
                                             CaptureFileName : String;
                                             CaptureBufferSize : Word);

```

Purpose

Create a CaptureTerminalWindow with custom options.

Description

This constructor instantiates a CaptureTerminalWindow using the specified WindowOptions and Colors. X1, Y1, X2, and Y2 are the coordinates of the interior portion of the window.

CaptureFileName is the name of the file to be used to store captured data. CaptureBufferSize specifies the size of the buffer used to store the data before it is flushed to disk.

InitCustom fails only if the capture buffer cannot be allocated or if the parent's constructor (TerminalWindow.InitCustom) fails.

See the entry for TerminalWindow.InitCustom for details concerning the meanings of the other parameters.

Example

```

var
  UP : UartPort;
  AE : AnsiEmulator;
  CTW : CaptureTerminalWindow;
...
UP.InitFast(Com1, 2400);
AE.Init(32);
if not CTW.InitCustom(
  1, 1, ScreenWidth, ScreenHeight, {window coordinates}
  DefaultColorSet,                  {colors}
  @UP,                              {port object}
  @AE,                              {terminal emulator}
  DefScrollBackRows,                {200 rows of scroll back}
  DefScrollBackCols,                { and 80 columns}
  DefTerminalWinOptions,            {terminal window options}
  'TEST.CAP',                       {capture file name}
  512) then                          {capture buffer size}
  {handle error} ;
...
CTW.Done;

```

Initialize a full-screen terminal window, communicating with the remote via Com1 at 2400 baud, with support for ANSI control sequences. The capture buffer will hold 512 characters at a time.

See Also

TerminalWindow.InitCustom

TerminalWindow.SetTerminalWinPort

TerminalWindow.SetTerminalWinEmulator

Load

Syntax

```
{IFDEF UseStreams}  
constructor CaptureTerminalWindow.Load(var S : IdStream);
```

Purpose

Load a CaptureTerminalWindow object from a stream.

Description

This constructor loads a CaptureTerminalWindow object from a stream.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to CaptureTerminalWindow.Store.

The addresses of both the terminal emulator object in use (if any) and the port object in use must be registered with the stream before Load is called.

The stream registration routine for a CaptureTerminalWindow object is CaptureTerminalWindowStream.

Example

```
uses  
  OpRoot, ApPort, OoCom, OpCrt, OpCmd, OpFrame, OpWindow, TermWin;  
var  
  UP : UartPort;  
  CTW : CaptureTerminalWindow;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not S.Init('TERMWIN.STM', SOpen, 1024) then begin  
    WriteLn('Error opening stream');  
    Halt;  
  end;  
  S.RegisterHier(CaptureTerminalWindowStream);  
  S.RegisterHier(UartPortStream);  
  S.RegisterPointer(1000, @UP);  
  S.Get(UP);  
  S.Get(CTW);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Load error: ', Status);  
    Halt;  
  end;  
  S.Done;  
  
  CTW.Process;  
  
  CTW.Done;  
  UP.Done;  
end.
```

Instantiates UP and CTW from a stream created by the Store example using the stream's Get method, which calls Load indirectly. As in the Store example, RegisterPointer is called to register the address of the port object prior to loading the terminal window, so that the link between the two can be reestablished.

Syntax

```
procedure CaptureTerminalWindow.SetCaptureFileName(FName : String);
```

Purpose

Set the name of the capture file.

Description

This routine allows you to change the name of the file used to capture incoming characters to FName. Note that if capture is already turned on, the current capture file is closed, and a new capture file is opened.

Example

```
CTW.SetCaptureFileName('MYPROG.LOG');
```

Use MYPROG.LOG as a capture file.

See Also

InitCustom

Store

Syntax

```
{IFDEF UseStreams}  
procedure CaptureTerminalWindow.Store(var S : IdStream);
```

Purpose

Store a CaptureTerminalWindow object to a stream.

Description

Store writes an image of the CaptureTerminalWindow object to the stream S.

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

The addresses of both the terminal emulator object (if any) and the port object must be registered with the stream before Store is called.

The stream registration routine for a CaptureTerminalWindow object is CaptureTerminalWindowStream.

Example

```
uses OpRoot, ApPort, OoCom, OpCrt, OpCmd, OpFrame, OpWindow, TermWin;  
var  
  UP : UartPort;  
  CTW : CaptureTerminalWindow;  
  S : BufIdStream;  
  Status : Word;  
begin  
  if not UP.InitFast(Com1, 2400) then begin  
    WriteLn('Error initializing port');  
    Halt;  
  end;  
  if not CTW.Init(2, 2, ScreenWidth-1, ScreenHeight-1, @UP) then begin  
    WriteLn('Error initializing terminal window');  
    Halt;  
  end;  
  CTW.WOptionsOn(wBordered);  
  if not S.Init('TERMWIN.STM', SCreate, 1024) then begin  
    WriteLn('Error initializing stream');  
    Halt;  
  end;  
  S.RegisterHier(CaptureTerminalWindowStream);  
  S.RegisterHier(UartPortStream);  
  S.RegisterPointer(1000, @UP);  
  S.Put(UP);  
  S.Put(CTW);  
  Status := S.GetStatus;  
  if Status <> 0 then begin  
    WriteLn('Store error: ', Status);  
    Halt;  
  end;  
  S.Done;  
  
  CTW.Process;  
  
  CTW.Done;  
  UP.Done;  
end.
```

Instantiates UP and CTW, and then stores both objects in a stream using the stream's Put method, which calls Store indirectly. RegisterPointer is called to register the address of the port object

prior to storing the terminal window, so that the link between the two can be reestablished when the objects are reloaded.

See Load for an example program that reloads these objects.

See Also

Load

TerminalWindow.Store

F. Demonstration Programs

Two example programs illustrate the use of the TERMWIN unit. The first, OOPCOM, is discussed in §1.F. The second, TERMTEST, is described here.

1. Terminal Window Test: TERMTEST

Much like SIMPCOM, TERMTEST is a fairly simple communications program written with the object oriented units in Async Professional. The biggest difference between them is that TERMTEST also builds on the user interface design tools in Object Professional and the TerminalWindow object implemented in TERMWIN.

Although its user interface is fancier than SIMPCOM's, the program itself does quite a bit less. It does not, for example, provide any means for uploading or downloading files. It illustrates how to use the TerminalWindow object and, equally important, how to derive new objects from it to extend its functionality.

TERMTEST expects a command line of the following form:

TERMTEST [Options]

The following options are available:

/Cportnum	This option specifies the com port that you want TERMTEST to use.
/Ffilename	This option tells TERMTEST to feed the contents of the specified file into the program's terminal window. When /F is specified, characters are taken from the file instead of the com port, effectively disabling the com port. This option is particularly useful for examining the contents of capture files that contain ANSI escape sequences.
/Sfilename	This option tells TERMTEST to store the objects it uses in the specified stream when the program ends, allowing you to save configuration information (window position/zoom state, com port, baud rate, etc.).
/Lfilename	This option tells TERMTEST to load the objects it uses from the specified stream, previously created by using the /S option.
/?	Display help screen.

Note that '-' can be used instead of '/' when specifying command line options.

Here are some examples of TERMTEST command line options:

TERMTEST /C2 /Stermtest.stm

Use Com2 and store objects in TERMTEST.STM when finished.

TERMTEST /Ltermtest.stm

Reload objects stored in TERMTEST.STM

TERMTEST /Fcapture.log

Display captured data stored in CAPTURE.LOG.

Once TERMTEST is loaded, the following commands are available in addition to the standard TerminalWindow commands:

<F1>

Help for TERMTEST.

<F2>

Toggle output wrapping. This command toggles the terminal window's `twWrapInWin` option, which is On by default.

<F3>

Dial modem. This command prompts you for a phone number to dial. To abort the operation, press `<Esc>`. If a phone number is specified and a connection is made, incoming text will appear in the terminal window.

<F4>

Clear the terminal window. Note that this command clears the entire virtual screen, not just that portion of it that is currently visible.

<F5>

Zoom/unzoom the terminal window. When zoomed, the window occupies the entire screen, not including the top and bottom lines, which are used to display the name of the program and status information.

<AltF5>

Maximize/unmaximize the terminal window. This command is similar to zoom/unzoom, except that when maximized the window retains its border, scrollbars, and mouse hot spots.

<F6>

When data is being fed into the terminal window from a file specified with the `/F` command line option, this command temporarily stops data from being fed into the terminal window. Issuing the command a second time resumes the process.

<F7>

Select next baud rate. This option allows you to select a baud rate by toggling through the list of available choices. The current baud rate setting appears on the status line at the bottom of the screen.

<F8>

Toggle between 8 data bits/no parity/1 stop bit (8N1) and 7 data bits/even parity/1 stop bit (7E1).

<F9>

When data is being fed in from a file, this command prompts you for the name of a new file to feed. To cancel the command, press `<Esc>`. When data is not being fed in from a file, this command does nothing.

<F10>

When data is being fed in from a file, this command tells TERMTEST to go back to the beginning of the file and display the data again.

<AltF1>

Toggle "floating view." When enabled, the viewport is automatically adjusted to keep incoming text visible in the viewport.

<AltF2>

This command causes TERMTEST to prompt you for the name of a file where it will store the contents of its virtual screen. To cancel the command, press `<Esc>`. This feature is intended to illustrate the use of TerminalWindow's `CopyVirtToFile` method.

<AltF3>

Exit from TERMTEST.

<AltF4>

This command causes TERMTEST to temporarily display a small window that overlaps the terminal window, illustrating TERMTEST's ability to continue receiving and displaying incoming text even when the terminal window isn't the current window. The overlapping window will disappear when a key or mouse button is pressed.

<AltC>

Toggle file capture. When you first issue this command, you are asked for the name of the file to store captured data in. If you issued this command by mistake, press <Esc>. Otherwise, TERMTEST creates the specified file (overwriting any existing file of the same name) and begins storing incoming text in it. If capture is already enabled, this command turns it off.

In general, TERMTEST is a far less useful program than either SIMPCOM or OOPCOM. But that is by design. TERMTEST isn't intended to tempt you to throw away your favorite communications program, but rather to illustrate the use of the TerminalWindow object and how to enhance it to meet specific needs.

10. Fax Support

A. Overview

Document transfer using facsimile (fax) machines has become quite common—you might even say pervasive—in today's business environment. A more recent development in fax devices is the "faxmodem." A faxmodem is a standard data modem that also has the ability, when used with appropriate software, to send and receive faxes. Since the faxmodem is under program control, it can provide more sophisticated capabilities than a dedicated fax machine. A few of the possibilities include database storage, editing, and forwarding of received documents, as well as scheduled fax transmissions to multiple recipients.

Unfortunately, controlling a faxmodem is a relatively complex task. As is typical for the communications industry, faxmodem technology is governed by multiple, evolving, incomplete standards. With the exception of the terse technical specifications offered by the TIA/EIA committee that controls faxmodem standards, little has been written about controlling faxmodems. The TIA/EIA specifications describe the bare necessities of faxmodem behavior, many ambiguities must be resolved by research and experimentation.

Async Professional's faxmodem units overcome this information shortage by providing a complete set of routines for computer control of faxmodems. These routines cover all phases of faxmodem usage including document conversion, send/receive support for the current generation of faxmodems, and support for the most popular "high-level" faxmodem manager: Intel/DCA's CAS (Communicating Applications Specification) modem manager software. These routines have a structure similar to Async Professional's file transfer protocols, complete with the many programming hooks that allow you to write full-featured applications.

Integrating faxmodem support into your application involves two central tasks:

- 1) document conversion
- 2) faxmodem send/receive

Document conversion means converting the document you want to transmit into a format suitable for fax transmission. Faxmodem send/receive covers all the steps needed to control a faxmodem when sending and receiving fax documents. These topics are discussed in more detail in the following sections.

Document Conversion

Document conversion is the process of creating a compressed bitmap image suitable for fax transmission. Async Professional can convert plain text files, PCX image files, and TIFF image files into fax file format. Files that have been converted to fax format in this way, as well as files that have been received by Async Professional's fax routines, are given the extension APF (Async Professional Fax).

Document conversion also includes the ability to display or print received fax files, and to convert them into another standard graphics format for further manipulation.

The APFAXCVT/OOFAXCVT units contain Async Professional's document conversion routines. When you convert a text file to a fax, you can use a built-in bitmapped font, or you can use a standard Hewlett-Packard LaserJet bitmapped font file. You can also convert monochrome PCX or TIFF files to APF format.

APFAXCVT/OOFAXCVT also provide a general routine that unpacks each compressed line of a received fax image into a pixel array, which is then passed to an output routine. One supplied

output routine writes out the pixel data in PCX file format, for display, editing, or further conversion by any of the PCX utilities that are widely available. Demonstration programs are also provided to display the unpacked images on VGA monitors and print them on Epson or H-P LaserJet compatible printers.

Faxmodem Send/Receive

Faxmodem send/receive is the process of sending the appropriate commands to a faxmodem to prepare it for sending or receiving faxes, initiating or receiving a call, transferring a bitmap image, and terminating the connection.

Like a file transfer protocol, a successful fax transmission requires cooperation between the sender and receiver. A number of standards have been developed for this purpose. Early fax machines communicated using what was known as the Group 1 and Group 2 protocols. Although your fax machine may have a Group 1 button, which allows it to receive faxes (very slowly) from an old Group 1 fax machine, all fax machines sold today support the Group 3 facsimile protocol. The faxmodems that Async Professional can control do not support Group 1 or Group 2 at all. If you need to send or receive faxes with a Group 1 partner, keep your old fax machine!

Within Group 3, there are currently two EIA/TIA standards for computer control of faxmodems: Class I and Class II. To identify these classes, Async Professional uses arabic numerals (Class 1 and Class 2) rather than roman numerals because the numbers are clearer to read in the source code and documentation. Class 2 depends on somewhat more sophisticated chips within the faxmodem than Class 1. Class 2 chips are capable of negotiating certain fax transmission parameters without any feedback from the computer.

There are also several standards that define a higher level of modem control than either Class 1 or Class 2. Faxmodem drivers that implement these standards provide a set of simpler functions to control the faxmodem: "send file" or "start receive mode," for example. Programs call these functions and the driver handles the details of converting the document and controlling the Class 1 or Class 2 modem hardware.

Although this approach is much simpler than controlling the faxmodem directly, it suffers from several problems. First, there is as yet no universal standard for the driver interface. At least three standards have been proposed: CAS, FaxBios, and Class 3. At this point CAS from Intel/DCA seems to be the most popular, but WordPerfect is putting its support behind FaxBios and the EIA/TIA committee itself is defining the Class 3 standard. Second, before running your application you must load another TSR or device driver that converts your high level calls into the low-level commands needed by the faxmodem itself. Distributing and installing such TSRs with an application is always prone to problems. And third, your application has less control over and visibility into the process of converting, sending, and receiving faxes.

In spite of these issues, Async Professional provides routines for controlling CAS-compatible faxmodems through the CAS manager TSR. It also provides routines for controlling Class 1 and Class 2 faxmodems directly.

The APABSFAX/OOABSFAX units provide "abstract" data structures, routines, and base objects for controlling faxmodems in general. This code is shared by other units that support specific types of faxmodems.

The APFAX12/OOFAX12 units provide routines for controlling Class 1 and Class 2 faxmodems. These two standards should cover any faxmodem you're likely to encounter today.

The APFAXCAS/OOFAXCAS units provide routines for controlling faxmodems that are using CAS faxmodem management software. Although the use of CAS is distinctly different from

controlling the faxmodem directly, the APFAXCAS/OOFAXCAS routines have been made as similar as possible to the routines in APFAX12/OOFAX12.

For Additional Information

Wherever possible the faxmodem support units insulate you from the details of document conversion and faxmodem control. Just as with the file transfer protocols, you don't need to read and understand all of the faxmodem technical specifications to use the faxmodem routines. However, it does help to have a basic understanding of the specifications when you are using the routines that are described in the following sections. For further information, obtain the following technical specifications:

RS-465	Group 3 Facsimile Apparatus for Document Transmission
RS-466	Procedures for Document Facsimile Transmission
EIA/TIA-578	Asynchronous Facsimile DCE Control Standard
EIA/TIA-592	Asynchronous Facsimile DCE Control Standard - Service Class II
DCA/Intel CAS	DCA/Intel Communicating Applications Specification

These documents are available from the EIA and TIA organizations directly. They can also be obtained from Global Engineering Documents, a company that distributes engineering specifications of all kinds. You can reach this company by telephone at 800-854-7179 or 714-261-1455. The CAS specification is available on the INTEL forum of CompuServe. The current file name is CAS12.COM for version 1.2 of the specification. The file name may change as the specification is revised.

B. OOP vs. non-OOP Interface

Async Professional's fax add-on layer continues the split started back at the interface layer. OOP and non-OOP routines are completely separate and have separate source code files: all OOP files have a prefix of OO- (e.g., OOFAX12) and all non-OOP files have a prefix of AP- (e.g., APFAX12).

Just like the interface layer, the routines in the fax add-on layer are nearly identical in the OOP and non-OOP formats. So again, they are documented only once, showing the appropriate declarations and pointing out the few differences. In the interface layer, the only consistent difference between the two formats was that the non-OOP format required a PortRecPtr. Similarly, the major difference between the two fax routine formats is that the non-OOP format requires a FaxRecPtr.

The OOP and non-OOP fax routines differ just a bit more in the area of procedure names. The OOP routines, by virtue of polymorphism, can use the same procedure names to initialize, process, and destroy all object types. The non-OOP routines can't do this.

A table illustrates this difference:

Purpose	OOP name	non-OOP name
initialize	Init	InitZzzConverter (e.g., InitTextConverter)
	Init	InitXxxYyy (e.g., InitC12SendFax)
convert	ConvertFax	ConvertFaxXxx (e.g., ConvertFaxText)
transmit	FaxTransmit	FaxTransmitXxx (e.g., FaxTransmitC12)
receive	FaxReceive	FaxReceiveXxx (e.g., FaxReceiveC12)
dispose	Done	DoneZzzConverter (e.g., DoneTextConverter)
	Done	DoneXxxYyy (e.g., DoneC12SendFax)

The OOP case is very simple. When converting faxes you always use Init, ConvertFax, Done; when sending faxes you always use Init, FaxTransmit, Done.

The non-OOP case is a bit more complex, but once understood, it too is very simple. The procedure names incorporate additional information that specifies the exact circumstances in which they should be used. In the table above, Zzz refers to the type of document conversion that should be performed: Text, Pcx, or Tiff. Xxx refers to the type of faxmodem that is being used for fax send or receive: C12 (Class 1 or 2) or CAS. And Yyy refers to the type of transmission: SendFax or ReceiveFax.

Let's look at some code fragments to get another view of the differences between OOP and non-OOP programs.

OOP:

```
uses
  ApMisc, ApPort, ApUart, OoCom, OoAbsFax, OoFax12;
var
  ComPort : UartPortPtr;
  Fax : C12SendFaxPtr;
...
  {Instantiate a port}
  New(ComPort, InitFast(Com3, 19200));
  {...handle errors...}

  {Instantiate a fax object}
  New(Fax, Init('TEST', ComPort));
  {...handle errors...}
```

```

{Transmit a fax}
Fax^.AddFaxEntry('260-7151', 'FAX.APF', '');
Fax^.FaxTransmit;

```

non-OOP:

```

uses
  ApMisc, ApPort, ApUart, ApCom, ApAbsFax, ApFax12;
var
  ComPort : PortRecPtr;
  Fax : FaxRecPtr;
...
{Initialize a port}
InitPortFast(ComPort, Com3, 19200);
{...handle errors...}

{Initialize a fax}
InitC12SendFax(Fax, 'TEST', ComPort);
{...handle errors...}

AddFaxEntry(Fax, '260-7151', 'FAX.APF', '');
FaxTransmitC12(Fax);

```

The OOP example declares a `UartPortPtr` (`ComPort`) and a `C12SendFaxPtr` (`Fax`). `ComPort` is instantiated first and is passed to `Fax`'s constructor. The example then goes on to transmit a document named `FAX.APF`.

The non-OOP example declares a `PortRecPtr` (`ComPort`) and a `FaxRecPtr` (`Fax`). `ComPort` is initialized first by the call to `InitPortFast`. The `ComPort` pointer is passed to the `InitC12SendFax` routine, which returns an initialized `Fax` pointer. The `Fax` pointer is then passed to all fax routines, as shown in the calls to `AddFaxEntry` and `FaxTransmitC12`.

The differences between the OOP and non-OOP format aren't inherently important, because you'll probably stick to one format only. However, understanding the differences will make the reference section and example programs much easier to follow.

One object hierarchy is used for converting documents to APF format, and another is used to convert from APF format. The non-OOP routines follow a similar design, but with record pointers instead of objects:

```

AbstractFaxConverter
├── TextFaxConverter
├── PcxFaxConverter
└── TiffFaxConverter

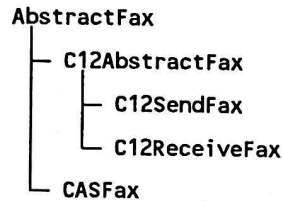
UnpackFax
└── UnpackToPcx

```

The `AbstractFaxConverter` contains the basic fields and methods that all converters need. One of the primary responsibilities of the `AbstractFaxConverter` is converting graphics raster lines (of bitmapped pixels) to a compressed ready-to-fax raster line. The `TextFaxConverter` reads ASCII files and uses a bitmapped font to convert each character to a bit image before creating the compressed raster lines. The `PcxFaxConverter` and `TiffFaxConverter` unpack PCX and TIFF monochrome graphics files and convert each graphics raster line into a fax image raster line.

The UnpackFax object works with both converted and received APF files. It decompresses each raster line and passes the bitmapped pixel array to a function or virtual method in your application. Your application can then print or display the data. The UnpackToPcx object overrides the same virtual method and provides additional utility methods to convert APF files directly into PCX format.

Another object hierarchy exists for the send and receive portions of the library. The non-OOP routines follow a similar design, but with record pointers instead of objects:



AbstractFax provides the low-level data and functions required of all fax devices. Class 1 and 2 faxmodems need a few fields and functions that aren't required for CAS faxmodems, hence the middle level abstract object C12AbstractFax. Since many applications don't require both send and receive capabilities, there are separate objects for send and receive.

C. Document Conversion: APFAXCVT/OOFAXCVT

Faxmodems don't transmit documents directly. Instead they transmit a compressed bitmap image in a format that is specific to Group 3 fax devices. APFAXCVT/OOFAXCVT provide objects and routines for converting documents into this format. They also provide routines for unpacking these compressed images for display, printing, or conversion to other graphics formats.

Throughout this chapter, the term "converting" is used to describe the process of transforming document files into fax format; the term "unpacking" describes the opposite transformation.

The Group 3 compressed bitmap format was designed specifically for transmission of data over possibly noisy lines. The compression technique results in a file that combines relatively small size with ease of recovery from missing or garbled data. As the receiving fax machine or faxmodem software detects bad compression strings, it simply discards them and starts collecting valid strings again. No transmission time is spent calculating and sending block check characters. A document received with line errors usually is missing just a few pixels; if the received document looks bad enough, the sender must transmit again.

Although the format of each compressed raster line is defined by the Group 3 facsimile specification, the APF file format used by Async Professional is proprietary. The data within each raster line follows the Group 3 spec, but the file contains additional information that makes it easier to manage and transmit the image.

An APF file is formatted as follows:

fax header
page header 1
page data
.
.
.
page header N
page data N

You usually don't need to understand this format in any detail, but the information is documented here in case you need to write APF manipulation utilities that aren't provided with Async Professional.

The file always begins with a header that contains, among other information, the number of pages in the fax. Each page of the document follows, including a page header and the compressed page data. The fax header and page header structures are defined in detail under "Shared Declarations" later in this section.

The page data is a series of compressed raster line images. The line image optionally begins with a word containing the number of bytes in the compressed line. (This word is stored only in APF files that are ready for transmission; it is used to aid in padding each line to match timing parameters of the receiving fax machine.) The length word is followed by the line image in Group 3 compression format. The raster line corresponds to a single horizontal row of bitmapped pixels in the original image. If a pixel is set, it corresponds to a black dot on the original image; if it is clear, it corresponds to a white dot.

The APF unpacking routines deliver one uncompressed raster line at a time to your program. These routines seek to the appropriate document page in the APF file, read the compressed line into memory, decompress it, pass the pixel image to an "OutputLine routine," then loop to get the next raster line on the page. An OutputLine routine can display the pixel image, print it, or convert it to another format before returning to the unpacking routine. The demonstration programs SHOWFAX/SHOWFAXO, PRNFAX/PRNFAXO, and FAX2PCX/FAX2PCXO illustrate each of these processes, respectively. OutputLine routines are discussed in "The OutputLine Hook," which appears later in this section.

Fax images can be converted and stored using two different resolutions. Standard resolution is 200 horizontal dots per inch by 100 vertical dots per inch. High resolution has the same horizontal resolution, but uses 200 vertical dots per inch. (In some fax documentation the resolutions are described as 98 dots per inch and/or 196 dots per inch. Those numbers are actually more exact, but 100 and 200 are easier to remember and are commonly used in most fax documentation.)

The standard width of a fax page is 1728 pixels or about 8.5 inches. Two optional widths are also available. Async Professional supports only one of the optional widths: 2048 pixels per row, or about 10 inches. There is no fundamental limit on fax page length. Even so, you'll probably want to limit it to 11 inches, or 14 inches if you want to mimic legal size paper. This is especially important when you consider that many faxes are now printed on sheet-fed laser printers.

You specify the fax resolution when the document is converted to an APF file. The resolution of each page is stored in its page header. When the APF file is later transmitted, the send fax object reads the page header to determine the resolution to use to transmit the fax.

Converting Documents

Converting a document to APF format is the first step in the fax send process. You can convert your documents just before you transmit them. If you like, this conversion process can be completely transparent to your users. Or, you can convert your documents in a separate step that is not immediately followed by transmission.

The following short programs are complete examples that show how to convert an ASCII text file to APF format. First the OOP version:

```

program ExCvt0; {EXCVT0.PAS}
uses
  Dos, Crt, ApMisc, OoFaxCvt;
var
  FCP : TextFaxConverterPtr;

{$F+}
function CvtStatusFunc(FCP : AbstractFaxConverterPtr;
  Starting, Ending : Boolean);
var
  Line : LongInt;
  Page : Integer;
begin
  FCP^.GetStatusInfo(Line, Page);
  Write('M'Processing line ', Line:4, ' of page ', Page:4);
  if KeyPressed then
    CvtStatusFunc := (ReadKey = #27)
  else
    CvtStatusFunc := False;
end;
{$F-}

```



```

begin
  FCP := New(TextFaxConverterPtr, Init);
  if FCP = nil then begin
    WriteLn('Error creating converter: ', AsyncStatus);
    Halt;
  end;

  {Set hook routine}
  FCP^.SetStatusFunc(CvtStatusFunc);

  {Convert it}
  FCP^.ConvertFax('EXCVTO.PAS');
  WriteLn('M^J'Conversion results: ', AsyncStatus);
  Dispose(FCP, Done);
end.

```

This program takes advantage of the optional conversion status hook. Any conversion program you write should use this hook so the user can see what's going on and cancel the conversion if necessary. This hook is discussed more fully later in this section.

The program instantiates a TextFaxConverter object (FCP) and assures that the object was successfully created. It then activates the hook routine mentioned above. It calls ConvertFax and passes it the name of the ASCII text file to convert, in this case the source file for this example. All of the work of the conversion is handled by ConvertFax, which reads EXCVTO.PAS line by line, rasterizes each line, and writes the compressed raster data and the appropriate fax and page headers to EXCVTO.APF. AsyncStatus contains the result of the conversion; it equals ecOk if the conversion was successful and contains a non-zero error code otherwise.

Here's the non-OOP version of the same program:

```

program ExCvt; {EXCVT.PAS}
uses
  Dos, Crt, ApMisc, ApFaxCvt;
var
  FC : FaxConverterPtr;

  {$F+}
  function CvtStatusFunc(FC : FaxConverterPtr;
    Starting, Ending : Boolean) : Boolean;
  var
    Line : LongInt;
    Page : Integer;
  begin
    GetStatusInfo(FC, Line, Page);
    Write('M^Processing line ', Line:4, ' of page ', Page:4);
    if KeyPressed then
      CvtStatusFunc := (ReadKey = #27)
    else
      CvtStatusFunc := False;
  end;
  {$F-}

begin
  InitTextConverter(FC);
  if FC = nil then begin
    WriteLn('Error initializing converter: ', AsyncStatus);
    Halt;
  end;
end;

```

```

{Set hook routine}
SetStatusFunc(FC, CvtStatusFunc);

{Convert it}
ConvertFaxText(FC, 'EXCVT.PAS');
WriteLn('M`J'Conversion results: ', AsyncStatus);
DoneTextConverter(FC);
end.

```

This program is identical in function to the OOP version. Of course, it uses the non-OOP calling conventions when creating the converter, setting the status hook, and converting the document.

The CVT2FAX/ CVT2FAXO demonstration programs provide more extensive examples of converting files to APF format.

ASCII Text Documents

The text converter provided in APFAXCVT/OOFAXCVT converts standard ASCII text files into APF files. Each text line of the input file must end with a carriage return and a line feed. The converter can handle all 256 characters in the standard IBM PC symbol set. APFAXCVT/OOFAXCVT cannot convert files that contain embedded word processor formatting commands.

The text converter reads each line of the text file and converts the line into an appropriate number of bitmapped raster lines. In essence, it converts each line into a picture of itself. In order to do this it uses a font table that contains a bitmap of each ASCII character. The number of raster lines per text line depends on the pixel height of each character and the vertical resolution of the fax conversion. The number of pixels in each raster line depends on the pixel width of each character. For each raster line and each character in the line, the converter finds the character bitmap in the font table and extracts the appropriate horizontal pixel row. After looping through all the rows in the font, the converter has created a bitmap image of the text line.

APFAXCVT/OOFAXCVT have two built-in fonts. SmallFont is a small 12x8 (12 pixels wide by 8 pixels high) font used for creating header lines at the top of each transmitted fax page. StandardFont is a 20x16 font used for all other text. StandardFont was chosen to provide text lines at 6 lines per inch or 66 lines on a standard 11 inch page. If you specify a high resolution image, the fonts are scaled vertically to 16 and 32 pixels, respectively.

The built-in fonts are stored in a file named APFAX.FNT, of size 16KB. This font file is required for converting text files and may be distributed with your applications. Alternatively, you can bind APFAX.FNT directly into APFAXCVT/OOFAXCVT by activating the compiler define BindFaxFont in APDEFINE.INC (it's undefined by default). When BindFaxFont is defined, APFAX.OBJ (created from APFAX.FNT using Turbo Pascal's BINOBJ utility) is linked into APFAXCVT/OOFAXCVT at compile time. This adds about 16KB of code to the unit and, ultimately, to your finished program.

As an alternative to the built-in fonts, you can use Hewlett-Packard LaserJet font files. Several restrictions limit the H-P font files you can use:

- 1) the font file must be for the LaserJet printer, not the DeskJet
- 2) the file must contain a bitmapped, not a scaleable, font
- 3) the file must not contain compressed character bitmaps

There are many commercial, shareware, and public domain font files that comply with these restrictions. Even so, you might not be pleased with the appearance of such fonts when they are used in a fax. The H-P character bitmaps are intended for a laser printer whose resolution (300 pixels per inch) is much higher than the resolution of a fax machine. When the font loading routine

scales the H-P font to the 20x16 format used internally, vital parts of the bitmap may be lost. Tests of various font files have produced the best results when H-P fonts with a point size between 6 and 10 are used.

See the LoadHPFont function for more information about using H-P font files.

There is another advantage to using H-P font files besides the different look you can obtain. When you use the fonts built into APFAXCVT/OOFAXCVT there is no support for international character sets. This support can be obtained by using H-P LaserJet font files that have an appropriate symbol set.

PCX/TIFF Graphic Images

Async Professional provides document conversion routines for two popular graphics image formats: PCX and TIFF. The PCX format originated with the PC Paintbrush program. The Async Professional conversion routines have been tested with PCX images up through version 3.0. TIFF (Tagged Image File Format) is a multi-platform format that is designed to allow easy migration between platforms such as the Macintosh and the IBM PC. The Async Professional conversion routines have been tested with TIFF images up through version 4.0.

The Async Professional conversion routines work with monochrome PCX and TIFF images only. A given PCX or TIFF file always produces an APF file containing one fax page.

Because PCX and TIFF images are already a sequence of compressed raster lines, the job of the APFAXCVT/OOFAXCVT routines is different than that performed by the text conversion routines. For PCX and TIFF files, the conversion routines unpack the PCX or TIFF images, then repack them into the format required for faxing.

PCX and TIFF images are stored assuming a pixel aspect ratio of 1 to 1, which means that the height of a pixel is the same as its width. For example, a 10 by 10 box of pixels would appear on screen as a square, not a rectangle. APF images, on the other hand, have one of two aspect ratios. In high resolution, the aspect ratio is 1 to 1. In standard resolution, the aspect ratio is 2 to 1. Hence, in a standard resolution fax, a 10 by 10 box of pixels would *not* appear as a square. Instead it would be a rectangle twice as high as it is wide.

Given the differences in aspect ratios, a PCX/TIFF image converted to a standard resolution APF image appears distorted—tall and thin. To solve this problem, APFAXCVT/OOFAXCVT double the width of an image whenever it is converted to a standard resolution fax and the doubled image still fits on the page. This behavior is enabled with the `fcDoubleWidth` option, which is on by default.

APFAXCVT/OOFAXCVT include another option, called `fcCenterImage`, that is used during PCX/TIFF conversions. When this option is enabled, as it is by default, graphic images are automatically centered horizontally on the fax page.

Unpacking APF Files

Fax images are transmitted and received in a compressed format. Hence, they must be unpacked before your program can view, print, or edit them.

The unpacking routines in APFAXCVT/OOFAXCVT handle all of the work of opening the APF file, finding the desired page, and uncompressing each raster line—but that's as far as the general purpose routines can go. The unpacked raster line is passed to your application through a procedure hook, which is responsible for printing, displaying, or otherwise processing the line.

Although APFAXCVT/OOFAXCVT won't print or display raster lines, demonstration programs are provided to show you how to do both. SHOWFAX/SHOWFAXO demonstrate how to display a graphics image on a VGA screen and scroll the image in response to keyboard commands. PRNFAX/PRNFAXO demonstrate how to print a bitmapped graphics image on Epson or LaserJet compatible printers.

APFAXCVT/OOFAXCVT automate one other operation that you're likely to perform on a received APF file: converting it to a different graphics format. These units support conversion only to the PCX format. A number of graphics image converters are available as public domain software, shareware, or commercial products. If you need image files in TIFF, GIF, BMP, or another format, use one of these utilities.

The OutputLine Hook

If the supplied unpacking functions or demonstrations don't meet the needs of your application, you can write your own OutputLine function. This is the hook called by the unpacker after it decompresses each raster line.

OutputLine must be a global, far function of the following form (OOP and non-OOP formats are the same):

```
{ $F+ }
function OutputLine(UP : UnpackFaxPtr; Buffer : PByteBuffer;
                   Len : Word; PH : PageHeaderRec) : Boolean;
begin
    ...
end;
```

The first parameter is a pointer to the UnpackFax object or record that called the OutputLine function. The second parameter, Buffer, is a pointer to a zero-based byte array containing the uncompressed raster data. Len is the length of the raster line in bytes. PH is a page header that you can use to determine the size of the image and whether it is in standard or high resolution. (See "Shared Declarations" below for details about the page header.)

OutputLine has one other role to play—it is also the abort hook for the unpacking process. If you want to abort the conversion process, your OutputLine function should return True; to continue the conversion it should return False.

Install an OutputLine hook by calling the SetOutputLineFunc procedure of APFAXCVT or OOFAXCVT. In OOFAXCVT, you can override the virtual method OutputLine instead.

The Conversion Status Hook

This hook is called regularly during conversion to inform the user of the progress of the conversion. The status routine must be a global, far function of the following form:

OOP:

```
{ $F+ }
function CvtStatusFunc(FCP : AbstractFaxConverterPtr;
                     Starting, Ending : Boolean) : Boolean;
begin
    ...
end;
```

non-OOP:

```
($F+)  
function CvtStatusFunc(FCP : FaxConverterPtr;  
                      Starting, Ending : Boolean) : Boolean;  
begin  
    ...  
end;
```

FCP is a pointer to the fax converter object or record. Starting is True only when the status routine is called the first time, giving you an opportunity to initialize window variables, clear the screen, etc. Ending is True only when the status routine is called the last time, allowing you to clean up the screen, dispose window buffers, etc.

Within the status routine, you may want to call the GetFileName and GetStatusInfo routines of APFAXCVT/OOFAXCVT. These routines return the name of the file and the current page and line being converted. Use the FCP parameter when calling these routines.

When a text file is being converted, the status routine is called once for each text line of the input file. When a PCX or TIFF file is being converted, the status routine is called for each UpdateInterval raster lines of input. UpdateInterval is an interfaced typed constant of APFAXCVT/OOFAXCVT that defaults to 16.

The status routine has one other role to play—it is also the abort hook for the conversion process. If you want to abort the conversion, your status function should return True; to continue the conversion it should return False.

Install a conversion status hook by calling the SetStatusFunc function of APFAXCVT or OOFAXCVT. In OOFAXCVT, you may override the virtual method FaxConvertStatus instead.

Shared Declarations

Constants

ApproFontName : PathStr = 'APFAX.FNT';

Name of the standard font file that is loaded by calling LoadFont. Modify this typed constant before calling LoadFont if the font file is not located in the current directory.

BadFaxCvtOptions = 0;

Conversion options for internal use, which cannot be changed by calling fcOptionsOn or fcOptionsOff. There are currently no such options.

BadUnpackOptions = 0;

Unpacking options for internal use, which cannot be changed by calling ufOptionsOn or ufOptionsOff. There are currently no such options.

DefFaxCvtOptions : Word = fcDoubleWidth+fcCenterImage;

Default fax conversion options.

DefUnpackOptions : Word = 0;

Default fax unpack options.

fcDoubleWidth = \$0001;

fcCenterImage = \$0002;

Options for fax conversion. These options affect PCX and TIFF conversions only. When fcDoubleWidth is set, as it is by default, the graphics image width is doubled to achieve the expected 1:1 aspect ratio. Doubling is automatically disabled when high resolution mode is set or the doubled image width will not fit onto the fax page. When fcCenterImage is set, as it is by default, graphics images are automatically centered horizontally on the fax page.

```

ffHighRes      = $0001;
ffHighWidth    = $0002;
ffLengthWords  = $0004;

```

Option flags for the fax page header. ffHighRes is set whenever the fax page is stored in high resolution format. ffHighWidth is set whenever the fax page depends on a long (2048 pixel) line. ffLengthWords is set whenever each compressed raster line in the page is preceded by a length word.

```

SmallFont = 16;
StandardFont = 48;

```

Font handles passed to the LoadFont function to identify the small or standard built-in font. The value of each constant equals the number of bytes per character in the font.

```

StandardWidth = 1728;
WideWidth     = 2048;

```

Fax pixel widths. 1728 pixels corresponds to 8.5 inch width; 2048 corresponds to 10 inch width.

```

ufHalfWidth = $0001;

```

Options for unpacking a fax. When ufHalfWidth is set, the unpacker automatically halves the number of pixels in each line.

```

UpdateInterval : Word = 16;

```

The number of raster lines between calls to the conversion status procedure, when converting PCX or TIFF files to fax format.

Types

```

FaxHeaderRec = record
  Signature : SigArray;           {APRO FAX signature}
  FDateTime : LongInt;           {Date and time in DOS format}
  SenderID   : String[20];       {Station ID}
  Filler     : Byte;            {Alignment byte, unused}
  PageCount  : Word;            {Number of pages in this file}
  PageOfs    : LongInt;         {Offset in file of first page}
  Padding    : Array[39..64] of Byte; {Expansion room}
end;

```

Defines the format of the header at the start of an APF file. Signature equals DefSig (APMISC). FDateTime is set to the date and time when the fax file was converted or received. SenderID is set to the string specified using SetStationID when the fax is converted, or to the transmitter's station ID when a fax is received. PageCount is the number of fax pages in this file. PageOfs is the offset into the file of the first page header. Padding fills the record out to 64 bytes, allowing room for additional information in the header.

```

PageHeaderRec = record
  ImgLength : LongInt;           {Bytes of image data in this page}
  ImgFlags   : Word;            {Image flags}
  Padding    : Array[7..16] of Byte; {Expansion room}
end;

```

Defines the format of the page header preceding each fax page in an APF file. ImgLength is the number of compressed bytes in this page. ImgFlags is a bitmapped word that can be interpreted using the ff- flags defined above. Padding fills the record out to 16 bytes, allowing room for additional information in the page header.

```

ResWidths = (rw1728, rw2048);

```

Acceptable fax widths. See SetResolutionWidth for details.

```

SigArray = Array[0..5] of Char;

```

Fax file signature array.

APFAXCVT Declarations

Types

```
FaxConverterPtr = ^FaxConverter;  
FaxConverter = record  
    ...  
end;
```

A generic FaxConverter record. A pointer of type FaxConverterPtr is initialized by the InitZzzConverter functions and must be passed to all non-OOP fax conversion routines. You should not need to refer to any of the fields of this record.

```
FaxCvtStatusFunc = function(FC : FaxConverterPtr;  
    Starting, Ending : Boolean) : Boolean;
```

A function of this type is called regularly during fax conversion to give you an opportunity to display status information or abort the conversion. See "The Conversion Status Hook" earlier in this section for more information.

```
OutputLineFunc = function(UFP : UnpackFaxPtr; Buffer : PByteBuffer;  
    Len : Word; PH : PageHeaderRec) : Boolean;
```

A function of this type is called during fax unpacking to process each uncompressed raster line. See "The OutputLine hook" earlier in this section for more information.

```
UnpackFaxPtr = ^UnpackFax;  
UnpackFax = record  
    ...  
end;
```

A generic UnpackFax record. A pointer of type UnpackFaxPtr is initialized by InitUnpacker and must be passed to all non-OOP fax unpacking routines. You should not need to refer to any of the fields of this record.

```
UnpackToPcxPtr = ^UnpackToPcx;  
UnpackToPcx = record  
    ...  
end;
```

An unpacking record used for fax to PCX conversion. A pointer of type UnpackToPcxPtr is initialized by InitUnpackToPcx and must be passed to UnpackFileToPcx and DoneUnpackToPcx.

OOFAXCVT Declarations

Types

```
AbstractFaxConverterPtr = ^AbstractFaxConverter;  
AbstractFaxConverter =  
    object(Root)  
    ...  
end;
```

Base fax converter object, used by the fax send engine directly, and also as an ancestor of the text, PCX, and TIFF fax converter objects. A pointer of type AbstractFaxConverterPtr is passed to the conversion user hooks, where it may be typecast if necessary to more specific types.

```
FaxCvtStatusFunc = function(AF : AbstractFaxConverterPtr;  
    Starting, Ending : Boolean);
```

A function of this type is called regularly during fax conversion to give you an opportunity to display status information or abort the conversion. See "The Conversion Status Hook" earlier in this section for more information.

```
OutputLineFunc = function(UFP : UnpackFaxPtr; Buffer : PByteBuffer;  
    Len : Word; PH : PageHeaderRec) : Boolean;
```

A function of this type is called during fax unpacking to process each uncompressed raster line. See "The OutputLine hook" earlier in this section for more information.

```

PcxFaxConverterPtr = ~PcxFaxConverter;
PcxFaxConverter =
    object(AbstractFaxConverter)
    ...
end;
Object for converting PCX files to fax format.
TextFaxConverterPtr = ~TextFaxConverter;
TextFaxConverter =
    object(AbstractFaxConverter)
    ...
end;
Object for converting text files to fax format.
TiffFaxConverterPtr = ~TiffFaxConverter;
TiffFaxConverter =
    object(AbstractFaxConverter)
    ...
end;
Object for converting TIFF files to fax format.
UnpackFaxPtr = ~UnpackFax;
UnpackFax =
    object(Root)
    ...
end;
Base object for unpacking fax files. You must call SetOutputLineFunc or override the virtual
method OutputLine to do anything with the unpacked output.
UnpackToPcxPtr = ~UnpackToPcx;
UnpackToPcx =
    object(UnpackFax)
    ...
end;
Object for converting fax files to PCX format.

```


Syntax

```
procedure ConvertFaxText(FC : FaxConverterPtr; FName : PathStr);
procedure ConvertFaxPcx(FC : FaxConverterPtr; FName : PathStr);
procedure ConvertFaxTiff(FC : FaxConverterPtr; FName : PathStr);
procedure TextFaxConverter.ConvertFax(FName : PathStr); virtual;
procedure PcxFaxConverter.ConvertFax(FName : PathStr); virtual;
procedure TiffFaxConverter.ConvertFax(FName : PathStr); virtual;
```

Purpose

Convert a file to APF format.

Description

The file to convert is specified by FName. A default file extension is applied according to the conversion type: 'TXT' for text, 'PCX' for PCX, and 'TIF' for TIFF.

The converted output file is named as follows. It is always given an extension of 'APF' (or the current value of the typed constant FaxFileExt from the APMISC unit). If SetFaxPath was called to specify a particular directory for converted files, the drive and directory are stripped from FName and the SetFaxPath drive and directory are added instead. If SetFaxPath was not called, the drive and directory of FName are left intact. If the output file already exists, it is overwritten without warning.

For text conversion, the ASCII text file is converted line by line, without any additional interpretation (of word processor commands, for example), using the font loaded by the last call to LoadFont or LoadHPFont. Each text line must end with a carriage return and a line feed character. Each rasterized line must be no longer than the current fax page width (default 1728 pixels, or about 85 StandardFont characters). Longer lines are truncated.

For PCX and TIFF conversion, note that ConvertFax accepts monochrome images only. The width of the image must not exceed the current fax page width (default 1728 pixels). If the image does not meet both of these conditions, ConvertFax returns immediately with AsyncStatus set to ecBadGraphicsFormat.

For TIFF conversion, ConvertFax converts only the first image of a multi-image file.

ConvertFax can take from several seconds to several minutes depending on the size of the input file and the speed of the machine. The conversion status hook is called once per text line for text files, or once per UpdateInterval (default 16) raster lines for PCX or TIFF files. If the status function returns True, ConvertFax deletes the partial output file and returns to the caller.

The status of the conversion is returned in the global variable AsyncStatus.

Examples

```
Converter^.ConvertFax('READ.ME');
if AsyncStatus = ecOk then
  WriteLn('Document successfully converted')
else
  WriteLn('Error during conversion: ', StatusStr(AsyncStatus));
```

OOP code that converts the text file READ.ME to the fax file READ.APF. AsyncStatus is checked to display the results of the conversion. The same OOP code would convert a PCX or TIFF file if the Converter object was declared and/or instantiated as the appropriate type.

```
ConvertFaxPcx(FC, 'PICTURE.PCX');
if AsyncStatus = ecOk then
  WriteLn('Document successfully converted')
else
  WriteLn('Error during conversion: ', StatusStr(AsyncStatus));
```

Non-OOP code for converting the PCX file PICTURE.PCX to the fax file PICTURE.APF.

See Also

LoadFont
LoadHPFont

SetFaxPath

Done / DoneZzzConverter

Syntax

```
procedure DoneTextConverter(var FC : FaxConverterPtr);
procedure DonePcxConverter(var FC : FaxConverterPtr);
procedure DoneTiffConverter(var FC : FaxConverterPtr);
destructor AbstractFaxConverter.Done; virtual;
destructor TextFaxConverter.Done; virtual;
destructor PcxFaxConverter.Done; virtual;
destructor TiffFaxConverter.Done; virtual;
```

Purpose

Dispose of a fax conversion object or record.

Description

These destructors and Done functions dispose of the converter object or record and any dynamic memory previously allocated by it.

You generally won't call AbstractFaxConverter.Done directly; it is called only by the destructors of objects derived from AbstractFaxConverter.

Example

See the examples for Init.

See Also

Init

Syntax

```
function fcOptionsAreOn(FC : FaxConverterPtr; OptionFlags : Word) : Boolean;  
function AbstractFaxConverter.fcOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose

Return True if all specified fax conversion options are on.

Description

This function returns True if the specified conversion options are currently selected.

Example

```
if fcOptionsAreOn(C, fcDoubleWidth) then  
    fcOptionsOff(C, fcDoubleWidth)  
else  
    fcOptionsOn(C, fcDoubleWidth);
```

Toggle the state of the fcDoubleWidth option. When using objects, the C parameters would be omitted.

See Also

fcOptionsOff

fcOptionsOn

Syntax

```
procedure fcOptionsOff(FC : FaxConverterPtr; OptionFlags : Word);  
procedure AbstractFaxConverter.fcOptionsOff(OptionFlags : Word);
```

Purpose

Deactivate multiple fax conversion options.

Description

This procedure deactivates the specified conversion options, excluding those designated as BadFaxCvtOptions.

Example

See the example for fcOptionsAreOn.

See Also

fcOptionsAreOn

fcOptionsOn

Syntax

```
procedure fcOptionsOn(FC : FaxConverterPtr; OptionFlags : Word);  
procedure AbstractFaxConverter.fcOptionsOn(OptionFlags : Word);
```

Purpose

Activate multiple fax conversion options.

Description

This procedure activates the specified conversion options, excluding those designated as BadFaxCvtOptions.

Example

See the example for fcOptionsAreOn.

See Also

fcOptionsAreOn

fcOptionsOff

GetFileName / GetStatusInfo

Syntax

```
function GetFileName(FC : FaxConverterPtr) : PathStr;  
function AbstractFaxConverter.GetFileName : PathStr;
```

Purpose

Return the name of the file being converted.

Description

This function is generally used in conversion status routines. It returns the same file name passed to the converter's initialization function, but always in upper case.

Example

```
...  
WriteLn('Converting ', GetFileName(C));  
...
```

Part of a status routine that displays the input file name. When using objects, the C parameter would be omitted.

See Also

SetStatusFunc

Syntax

```
procedure GetStatusInfo(FC : FaxConverterPtr; var Line : LongInt;  
                        var Page : Integer);  
procedure AbstractFaxConverter.GetStatusInfo(var Line : LongInt;  
                                             var Page : Integer);
```

Purpose

Return the current line number and page number.

Description

This procedure is generally used in conversion status routines to obtain the current page and line number. Pages are numbered starting at one. Lines are also numbered starting at one. For text files, the line number is incremented once per text line. For PCX and TIFF files, the line number is incremented for each UpdateInterval raster lines.

Example

```
...  
GetStatusInfo(C, Line, Page);  
Write('M'Processing line ', Line:4, ' of page ', Page:4);  
...
```

Part of a status procedure that writes the current line and page number each time it is called. When using objects, the C parameter would be omitted.

See Also

SetStatusFunc

Syntax

```
procedure InitTextConverter(var FC : FaxConverterPtr);
procedure InitPcxConverter(var FC : FaxConverterPtr);
procedure InitTiffConverter(var FC : FaxConverterPtr);
constructor AbstractFaxConverter.Init;
constructor TextFaxConverter.Init;
constructor PcxFaxConverter.Init;
constructor TiffFaxConverter.Init;
```

Purpose

Allocate and initialize a fax conversion object or record.

Description

These constructors and initialization functions initialize the data fields of a fax converter capable of performing the specified type of conversion. Each function allocates about 32KB of heap space needed for file buffers, compression buffers, and font tables.

Status information is returned in the global variable AsyncStatus. If the non-OOP initialization fails, the FC parameter is set to nil.

You generally won't call AbstractFaxConverter.Init directly; it is called only by the constructors of objects derived from AbstractFaxConverter.

Examples

```
var
  C : TextFaxConverterPtr;
begin
  New(C, Init);
  if C = nil then
    {...handle error...}
  C^.ConvertFax('READ.ME');
  Dispose(C, Done);
end.
```

A minimal object-oriented program that converts the text file READ.ME to the fax file READ.APF. The only change to convert a PCX or TIFF file would be to declare a variable of type PcxFaxConverterPtr or TiffFaxConverterPtr instead of TextFaxConverterPtr.

```
var
  C : FaxConverterPtr;
begin
  InitTextConverter(C);
  if C = nil then
    {...handle error...}
  ConvertFaxText(C, 'READ.ME');
  DoneTextConverter(C);
end.
```

A minimal non-OOP program that converts the text file READ.ME to the fax file READ.APF. The only change to convert a PCX or TIFF file would be to replace the string 'Text' in the function names with 'Pcx' or 'Tiff', respectively.

See Also

Done

LoadFont

Syntax

```
function LoadFont(FC : FaxConverterPtr; FontHandle : Byte;  
                  HiRes : Boolean) : Boolean;  
function AbstractFaxConverter.LoadFont(FontHandle : Byte;  
                                       HiRes : Boolean) : Boolean; virtual;
```

Purpose

Load a font from the default font file.

Description

This procedure loads a font from the supplied file APFAX.FNT. If APFAX.FNT is not in the current directory, be sure to update the typed constant AproFontName before calling LoadFont. If BindFaxFont is defined in APDEFINE.INC (it's undefined by default), LoadFont instead performs the necessary steps to activate the fonts that are already linked into the APFAXCVT/OOFAXCVT code segment. LoadFont cannot fail when BindFaxFont is defined. HiRes controls the resolution of the conversion: False for standard resolution, True for high resolution.

The two acceptable values for FontHandle are SmallFont and StandardFont. You should almost always choose StandardFont. This font is 20x16 for standard resolution and is scaled up to 20x32 for high resolution. In either resolution this font provides exactly 6 text lines per inch or 66 lines on a standard 11 inch page. There is little reason to use high resolution conversion for a text file using the built-in font, since you get exactly the same results as with standard resolution but at twice the transmission time.

SmallFont is a small 12x8 font used by APABSFAX/OOABSFAX for generating header lines at the top of each transmitted page. You can use it for the main text font if you want lots of small text on each fax page.

LoadFont returns False if it was unable to load the font. In this case, AsyncStatus holds an error code with an explanation of the failure.

Example

```
if not LoadFont(C, StandardFont, True) then begin  
    {...handle errors...}  
end;
```

Assuming BindFaxFont was undefined during compilation, this example loads StandardFont from APFAX.FNT and scales it up to high resolution format. The code must be prepared to handle errors.

```
if not LoadFont(C, StandardFont, False) then ;
```

Assuming BindFaxFont was defined during compilation, this example loads StandardFont in standard resolution from the bound font table. It's not necessary to check for errors in this case.

When using objects, the C parameters would be omitted.

See Also

LoadHPFont

Syntax

```
function LoadHPFont(FC : FaxConverterPtr; FontName : PathStr;  
                   HiRes : Boolean) : Boolean;  
  
function AbstractFaxConverter.LoadHPFont(FontName : PathStr;  
                                         HiRes : Boolean) : Boolean;
```

Purpose

Load a font from an H-P LaserJet font file.

Description

This procedure loads a requested font from an H-P LaserJet file named FontName. If FontName doesn't include a drive and directory name, the font file must be found in the current directory.

The H-P font file must adhere to several restrictions: it must be a LaserJet font, not a DeskJet font; it must be bitmapped, not scaleable; and the character bitmaps must not be compressed. If these requirements are not met, LoadHPFont returns False with AsyncStatus set to ecFontNotSupported.

LoadHPFont scales each character bitmap in the font file to 20x16 if HiRes is False, 20x32 if HiRes is True. H-P font files for large point sizes use bitmaps much larger than 20x32, which often leads to distorted characters after scaling. Best results are obtained by using point sizes between 6 and 10.

Example

```
if UseHPFont then begin  
  if not LoadHPFont(C, 'HV08R#US.SFP', False) then  
    {...handle errors...}  
end else begin  
  if not LoadFont(C, StandardFont, False) then  
    {...handle errors...}  
end;
```

Loads either an H-P font file or the built-in StandardFont. When using objects, the C parameters would be omitted.

See Also

LoadFont

SetFaxPath / SetPageSize / SetResolutionMode

Syntax

```
procedure SetFaxPath(FC : FaxConverterPtr; PS : PathStr);  
procedure AbstractFaxConverter.SetFaxPath(PS : PathStr);
```

Purpose

Set the path for storing fax files.

Description

This routine allows you to specify the drive:\directory (PS) for storing converted APF files. If only a drive is specified (e.g., 'D:'), the files are stored in the current directory for that drive. If SetFaxPath is not called, converted files are stored in the current directory.

Example

```
SetFaxPath(C, 'D:\OUTBOX');
```

Store converted files in D:\OUTBOX. When using objects, the C parameter would be omitted.

Syntax

```
procedure SetPageSize(FC : FaxConverterPtr; PS : Integer);  
procedure AbstractFaxConverter.SetPageSize(PS : Integer);
```

Purpose

Set the page size for text faxes.

Description

This routine is used to set the fax page length measured in text lines. After PS lines of text are processed, the current page is closed and a new page added to the APF file.

If PS is less than or equal to zero, no page breaks are inserted. This is the default behavior.

The page size does not apply to PCX or TIFF conversions, where the result of the conversion always defines one page.

Example

```
SetPageSize(C, 60);
```

Sets the page size to 60 text lines per page for text converters. When using objects, the C parameter would be omitted.

Syntax

```
procedure SetResolutionMode(FC : FaxConverterPtr; HiRes : Boolean);  
procedure AbstractFaxConverter.SetResolutionMode(HiRes : Boolean);
```

Purpose

Select standard or high resolution mode.

Description

If HiRes is True, the converted document is stored with 200 horizontal by 200 vertical pixels per inch. If HiRes is False, the document is stored with 200 horizontal by 100 vertical pixels per inch. The default resolution is HiRes = False, or 100 vertical pixels per inch.

If you are performing a text conversion, the HiRes parameter passed to LoadFont or LoadHPFont must match the value passed to SetResolutionMode.

Example

```
SetResolutionMode(C, UseHiRes);
```

Sets the resolution mode to UseHiRes, assumed to be a boolean flag previously set by a command line parameter. When using objects, the C parameter would be omitted.

Syntax

```
procedure SetResolutionWidth(FC : FaxConverterPtr; RW : ResWidths);  
procedure AbstractFaxConverter.SetResolutionWidth(RW : ResWidths);
```

Purpose

Select the resolution width for a fax.

Description

This routine is used to select one of the page widths supported by Async Professional. Acceptable values for RW are rw1728 (standard width: 1728 pixels or about 8.5 inches) and rw2048 (extended width: 2048 pixels or about 10 inches).

The default width is 1728.

Example

```
SetResolutionWidth(C, rw2048);
```

Selects the optional 2048 pixel width. When using objects, the C parameter would be omitted.

Syntax

```
procedure SetStationID(FC : FaxConverterPtr; SID : Str20);  
procedure AbstractFaxConverter.SetStationID(SID : Str20);
```

Purpose

Set the sending station ID.

Description

A fax device can identify itself to another fax device with a 20 character name, called the station ID. The Class 1 and Class 2 specifications indicate that the station ID should contain just a phone number, therefore they limit it to just the digits 0 through 9 and space. However, the station ID is frequently used to store an alphabetic name. Most faxmodems support this convention by allowing upper and lower case letters as well as other special characters in the station ID. This can cause problems for some fax machines, though, since they cannot print these characters.

Async Professional does not filter the characters stored in the station ID. You may wish to limit the characters passed to SetStationID in order to make your software compatible with the broadest possible range of fax hardware.

The SetStationID routines in APFAXCVT/OOFAXCVT control the station ID string that is stored in the header of the converted APF file. This string may or may not be used when the fax is transmitted, since separate SetStationID routines in APABSFAX/OOABSFAX set the ID string that is sent through the faxmodem. If you want to send the ID string embedded in the APF file, your software should read the SendID string from the fax header and pass it to APABSFAX/OOABSFAX before sending.

Example

```
SetStationID(C, 'MAIN OFFICE');
```

Embeds the station ID 'MAIN OFFICE' in the fax header of APF files. When using objects, the C parameter would be omitted.

See Also

SetStationID (APABSFAX)

AbstractFax.SetStationID

SetStatusFunc

Syntax

```
procedure SetStatusFunc(FC : FaxConverterPtr; FCF : FaxCvtStatusFunc);  
procedure AbstractFaxConverter.SetStatusFunc(FCF : FaxCvtStatusFunc);
```

Purpose

Activate a conversion status function.

Description

This routine allows you to specify a function that is called during fax file conversion to display status information and to check for a user abort of the conversion.

The function passed as FCP must be far and global, and must match the form of the FaxCvtStatusFunc type.

To deactivate a status function, call SetStatusFunc and pass nil as the FaxCvtStatusFunc.

See "The Conversion Status Hook" earlier in this section for more information and examples.

Example

```
SetStatusFunc(F, MyStatusFunc);
```

Activates MyStatusFunc as the status function. When using objects, the F parameter would be omitted.

Syntax

```
procedure DoneUnpacker(var UF : UnpackFaxPtr);  
procedure DoneUnpackToPcx(var UFP : UnpackToPcxPtr);  
destructor UnpackFax.Done; virtual;
```

Purpose

Dispose of an unpacker object or record.

Description

This destructor or Done procedure disposes of the unpacker object or record and any dynamic memory previously allocated by it.

In the non-OOP versions, the parameter is set to nil on return.

Note that UnpackFax.Done is inherited and used by the UnpackToPcx object as well.

Example

See the example for Init.

GetFaxHeader / GetPageHeader

Syntax

```
procedure GetFaxHeader(UF : UnpackFaxPtr; FName : PathStr;  
                      var FH : FaxHeaderRec);  
  
procedure UnpackFax.GetFaxHeader(FName : PathStr;  
                                var FH : FaxHeaderRec);
```

Purpose

Return an APF fax header.

Description

This procedure opens the APF file specified by FName, reads the fax file header into FH, and closes the file. You shouldn't generally need to call this routine, but it's provided just in case. For an example of when you might need it, see the entry for SetStationID.

Status information is returned in the global variable AsyncStatus.

Example

```
var  
  U : UnpackFaxPtr;  
  FH : FaxHeaderRec;  
...  
  GetFaxHeader(U, 'BIG.APF', FH);  
  if AsyncStatus <> ecOK then  
    {...handle error...}  
  else  
    WriteLn('This fax has ', FH.PageCount, ' pages.');
```

Reads the fax header of the BIG.APF fax file into FH. The PageCount field of FH is then displayed. When using objects, the U parameter would be omitted.

Syntax

```
procedure GetPageHeader(UF : UnpackFaxPtr; FName : PathStr; Page : Word;  
                       var PH : PageHeaderRec);  
  
procedure UnpackFax.GetPageHeader(FName : PathStr; Page : Word;  
                                  var PH : PageHeaderRec);
```

Purpose

Return the header for a page of a fax file.

Description

This procedure opens the file specified by FName, reads the page header specified of the given Page number into PH, and closes the file. You shouldn't generally need to call this routine, but it's provided just in case.

Status information is returned in the global variable AsyncStatus.

Example

```
var  
  U : UnpackFaxPtr;  
  PH : PageHeaderRec;  
...  
  GetPageHeader(U, 'BIG.APF', 2, PH);  
  if AsyncStatus <> ecOK then  
    {...handle error...}  
  else  
    WriteLn('This page has ', PH.ImgLength, ' bytes.');
```

Reads the page header of the second page of the BIG.APF fax file into PH. The ImgLength field of FH is then displayed. When using objects, the U parameter would be omitted.

Syntax

```
procedure InitUnpacker(var UF : UnpackFaxPtr);  
procedure InitUnpackToPcx(var UFP : UnpackToPcxPtr);  
constructor UnpackFax.Init;
```

Purpose

Allocate and initialize an unpacker object or record.

Description

These constructors and initialization procedures initialize an unpacker object or record. They allocate approximately 10KB of heap space for various buffers.

Status information is returned in the global variable AsyncStatus.

Note that UnpackFax.Init is inherited and used by the UnpackToPcx object as well.

Examples

```
U.Init;  
U.SetOutputLineFunc(MyOutputLine);  
U.UnpackFax('BIG.APF');  
U.Done;
```

Minimal OOP program that unpacks the file BIG.APF.

```
InitUnpacker(U);  
SetOutputLineFunc(U, MyOutputLine);  
UnpackFax(U, 'BIG.APF');  
DoneUnpackFax(U);
```

Minimal non-OOP program that unpacks the file BIG.APF.

```
InitUnpackToPcx(U);  
UnpackFaxToPcx(U, 'BIG.APF');  
DoneUnpackToPcx(U);
```

Minimal non-OOP program to unpack an APF file to a PCX file. This program does not need to specify an OutputLineFunc because UnpackFaxToPcx installs its own. You can also specify your own OutputLineFunc if you want to display status information during unpacking.

See Also

Done

SetOutputLineFunc

Syntax

```
procedure SetOutputLineFunc(UF : UnpackFaxPtr; OLF : OutputLineFunc);  
procedure UnpackFax.SetOutputLineFunc(OLF : OutputLineFunc);
```

Purpose

Activate an OutputLine function.

Description

This routine allows you to specify a function that is called each time UnpackFile or UnpackPage unpacks a raster line. A program that unpacks an APF file should *always* provide an OutputLineFunc or override the virtual method OutputLine. If you don't do so, and if you're not unpacking to create a PCX file, the unpack routines will produce no output.

The function passed as OLF must be far and global, and must match the form of the OutputLineFunc type.

See "The OutputLine Hook" earlier in this section for more information and examples.

Example

```
SetOutputLineFunc(U, MyOutputLine);
```

Sets MyOutputLine as the output procedure. When using objects, the U pointer would be omitted.

See Also

UnpackFile

UnpackPage

ufOptionsAreOn / ufOptionsOff / ufOptionsOn

Syntax

```
function ufOptionsAreOn(UF : UnpackFaxPtr; OptionFlags : Word) : Boolean;  
function UnpackFax.ufOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose

Return True if all specified unpacking options are on.

Description

This function returns True if the specified unpack options are currently selected.

Example

```
if ufOptionsAreOn(U, ufHalfWidth) then  
  ufOptionsOff(U, ufHalfWidth)  
else  
  ufOptionsOn(U, ufHalfWidth);
```

Toggle the state of the ufHalfWidth option. When using objects, the U parameters would be omitted.

See Also

ufOptionsOff

ufOptionsOn

Syntax

```
procedure ufOptionsOff(UF : UnpackFaxPtr; OptionFlags : Word);  
procedure UnpackFax.ufOptionsOff(OptionFlags : Word);
```

Purpose

Deactivate multiple unpacking options.

Description

This procedure deactivates the specified unpack options, excluding those designated as BadUnpackOptions.

Example

See the example for ufOptionsAreOn.

See Also

ufOptionsAreOn

ufOptionsOn

Syntax

```
procedure ufOptionsOn(UF : UnpackFaxPtr; OptionFlags : Word);  
procedure UnpackFax.ufOptionsOn(OptionFlags : Word);
```

Purpose

Activate multiple unpacking options.

Description

This procedure activates the specified unpack options, excluding those designated as BadUnpackOptions.

Example

See the example for ufOptionsAreOn.

See Also

ufOptionsAreOn

ufOptionsOff

UnpackFile / UnpackFileToPcx

Syntax

```
procedure UnpackFile(UF : UnpackFaxPtr; FName : PathStr);
procedure UnpackFileToPcx(UFP : UnpackToPcxPtr; Fax, Pcx: PathStr);
procedure UnpackFax.UnpackFile(FName : PathStr);
procedure UnpackToPcx.UnpackFileToPcx(Fax, Pcx: PathStr);
```

Purpose

Unpack all pages in a fax file.

Description

These routines unpack the entire APF file specified by FName. FName can include directory information. If it does not, FName is assumed to be in the current directory.

UnpackFile unpacks each raster line, passing the unpacked raster line to the OutputLine function. For UnpackFileToPcx, an internal OutputLine function is installed to convert each raster line to a PCX format raster line. If you install your own OutputLine function for UnpackFileToPcx, it will still be called, allowing you to display status or check for a user abort.

Status information is returned in the global variable AsyncStatus.

Examples

```
SetOutputLineFunc(U, MyOutputLine);
UnpackFile(U, 'BIG.APF');
```

Non-OOP code that unpacks each raster line of each page in BIG.APF and passes each line to MyOutputLine. When using objects, the U parameter would be omitted.

```
var
  U : UnpackToPcxPtr;
...
New(U, Init);
if U = nil then
  {...handle errors...}
U^.UnpackFileToPcx('BIG.APF', 'BIG.PCX');
if AsyncStatus <> ecOk then
  {...handle errors...}
Dispose(U, Done);
```

A minimal OOP program that creates a PCX file from the fax file BIG.APF.

See Also

SetOutputLineFunc

UnpackPage

Syntax

```
procedure UnpackPage(UF : UnpackFaxPtr; FName : PathStr; Page : Word);  
procedure UnpackFax.UnpackPage(FName : PathStr; Page : Word);
```

Purpose

Unpack a page.

Description

This routine unpacks just the specified Page of the APF file FName. The first page is number 1. FName can include directory information. If it does not, FName is assumed to be in the current directory.

UnpackPage unpacks each raster line in the page, passing the unpacked raster line to the OutputLine function.

This routine is not used for unpacking fax files to the PCX format, since such files are always treated as a single page.

Status information is returned in the global variable AsyncStatus.

Example

```
SetOutputLineFunc(U, MyOutputLine);  
UnpackPage(U, 'BIG.APF', 2);
```

Unpacks each raster line of page 2 in BIG.APF and passes each unpacked raster line to MyOutputLine. When using objects, the U parameter would be omitted.

See Also

SetOutputLineFunc

UnpackFile

D. Abstract Fax Send/Receive: APABSFAX/OOABSFAX

Async Professional's tools for sending and receiving faxes are organized into three groups of units. APABSFAX and OOABSFAX provide the "abstract" group, which defines various data types and functions that are needed regardless of the kind of faxmodem hardware. APFAX12 and OOFAX12, described in §10.E, build onto the abstract group to support Class 1 and Class 2 faxmodem hardware. APFAXCAS and OOFAXCAS, described in §10.F, also use the abstract group to support faxmodems that are using CAS faxmodem driver software.

APABSFAX/OOABSFAX define records and objects that include the name and size of the file being sent or received, the station ID of the sender or receiver, and the status of the current transfer. They also define the hooks that allow your application to log fax calls, generate file names for incoming faxes, report status during transmissions, and others. These capabilities, as well as a general overview of the send/receive process, are described in this section.

If you already know how to use file transfer protocols within the Async Professional architecture, fax transfers will seem very familiar. Just like a file transfer, a fax transfer involves opening a com port, initializing a fax transfer object or record, calling a function to send or receive the fax, and disposing of the transfer object. The user hooks for fax transfer are quite similar, and in some cases identical, to those for file transfer.

There are a few differences, though, and these differences are inherent to the nature of faxes. Unlike most file transfer protocols, where the name and size of the file are transmitted before the file's contents are sent, a fax receiver isn't given a filename in which to store an incoming fax, and it doesn't know the size of the fax until the transmission terminates. The fax protocol also provides no means for the receiver to detect errors in the incoming data, or to request retransmission in the event of bad data. Fortunately, the compression technique used for fax data is relatively tolerant of errors. Because of these differences, your program must take on certain responsibilities that are not required when using file transfer protocols.

The following simple example programs, based on the APFAX12/OOFAX12 units, show how easy it is to send a fax. First, the OOP version:

```
program ExSend0; {EXSEND0.PAS}
uses
  Dos, Crt, ApMisc, ApPort, ApUart, OoCom, OoAbsFax, OoFax12;
var
  Sender : C12SendFaxPtr;
  ComPort : UartPortPtr;
  Class1, Class2 : Boolean;

  {$F+}
  function FaxAbort : Boolean;
  begin
    if KeyPressed then
      FaxAbort := ReadKey = #27
    else
      FaxAbort := False;
  end;

  procedure FaxStatus(FP : AbstractFaxPtr; Starting, Ending : Boolean);
  begin
    Write('M' Transmit state: ', GetFaxProgress);
    ClrEol;
  end;
  {$F-}
```

```

begin
  ComPort := New(UartPortPtr, InitCustom(Com3, 19200, NoParity, 8, 1,
                                         4096, 4096, DefPortOptions));
  Sender := New(C12SendFaxPtr, Init('EXSEND0', ComPort));
  if (ComPort = nil) or (Sender = nil) then begin
    WriteLn('Unable to create port or fax, status: ', AsyncStatus);
    Halt;
  end;

  ComPort^.SetAbortFunc(FaxAbort);
  Sender^.SetFaxStatusProc(FaxStatus);

  Sender^.SetModemInit('&K5');

  if Sender^.GetModemClassSupport(Class1, Class2, True) then begin
    if not Class1 and not Class2 then begin
      WriteLn('Not a class 1 or class 2 modem');
      Halt;
    end;
  end else begin
    WriteLn('Failed to identify modem, status: ', AsyncStatus);
    Halt;
  end;

  Sender^.AddFaxEntry('260-7151', 'EXCVTO.APF', '');
  Sender^.FaxTransmit;
  WriteLn('M^J'Results of fax send: ', AsyncStatus);

  Dispose(Sender, Done);
  Dispose(ComPort, Done);
end.

```

This is a complete program for sending the fax file EXCVTO.APF, which was created by the example program EXCVTO.PAS described in §10.C.

The program first instantiates a port object (ComPort) and a fax send object (Sender). ComPort is opened at 19.2K baud to communicate between the PC and the faxmodem. The fax send object will later negotiate a different baud rate for the faxmodem-faxmodem connection.

The station ID of the sender, 'EXSEND0', is specified when Sender is instantiated. This string is sent to the receiving faxmodem as the fax connection is being made.

A standard port abort routine is activated with the call to SetAbortFunc. The FaxAbort function specified here aborts the fax transmission if <Esc> is pressed.

The fax status hook is enabled with the call to SetFaxStatusProc. The FaxStatus procedure specified here is called regularly during the fax transmission, providing an opportunity for the program to inform the user about what's going on. The simple example here just prints the value returned by GetFaxProgress. The status hook is described in more detail later in this section.

The program specifies '&K5' as the custom initialization string to be sent to the faxmodem. This string enables the faxmodem's software flow control option, which may be necessary to account for the difference in transfer rates between the PC-faxmodem connection (19.2K baud) and the faxmodem-to-faxmodem connection (14400 BPS or less). Some faxmodems do not require such custom initialization, since they automatically enable software flow control when sending or

receiving faxes. In addition, the command used to enable flow control varies among faxmodems. See the file MODEM.DOC in your \APRO directory for details about specific modems.

The call to GetModemClassSupport attempts to classify the sender's faxmodem by sending it a query string and getting the modem's response. This program requires a faxmodem attached to Com3 that supports the Class 1 or Class 2 specification. If this requirement isn't met, the program halts.

With this preparation out of the way, the program is ready to send a fax. The AddFaxEntry method specifies the receiver's phone number (260-7151) and the fax file to send (EXCVTO.APF). The third parameter to AddFaxEntry, not used here, specifies a file that is used as a fax cover page. Additional calls to AddFaxEntry would queue additional faxes to be sent.

All of the work of transmitting the fax is handled within the call to FaxTransmit. It opens the fax file, dials the phone number, handshakes with the remote faxmodem, sends the file, and disconnects. During the transmission, FaxTransmit calls the user hook routines, temporarily returning control to the example program to update the status line and check for a keyboard abort. If additional faxes were queued, FaxTransmit would stay in control until all of them were sent or an error occurred. Only then would FaxTransmit return to the example program.

The program then checks AsyncStatus to see whether the transmission was successful. Even though the fax protocol does not detect data transmission errors, there are plenty of ways a transmission can fail. For example, there might be no answer at the other end, the answerer might not be a compatible faxmodem, the connection might be broken in the middle of transmission, or the receiver might run out of paper or disk space.

The program then disposes of the fax send object and the port object.

Here's the non-OOP version of the same program:

```
program ExSend; {EXSEND.PAS}
uses
  Dos, Crt, ApMisc, ApPort, ApUart, ApCom, ApFaxCvt, ApAbsFax, ApFax12;
var
  Sender : FaxRecPtr;
  ComPort : PortRecPtr;
  Class1, Class2 : Boolean;

  {$F+}
  function FaxAbort : Boolean;
  begin
    if KeyPressed then
      FaxAbort := ReadKey = #27
    else
      FaxAbort := False;
  end;

  procedure FaxStatus(FP : FaxRecPtr; Starting, Ending : Boolean);
  begin
    Write('MTransmit state: ', GetFaxProgress);
    ClrEol;
  end;
  {$F-}

begin
  InitPort(ComPort, Com3, 19200, NoParity, 8, 1,
    4096, 4096, DefPortOptions);
```

```

InitC12SendFax(Sender, 'EXSEND', ComPort);
if (ComPort = nil) or (Sender = nil) then begin
  WriteLn('Unable to create port or fax, status: ', AsyncStatus);
  Halt;
end;

SetAbortFunc(ComPort, FaxAbort);
SetFaxStatusProc(Sender, FaxStatus);

SetModemInit(Sender, '&K5');

if GetModemClassSupport(Sender, Class1, Class2, True) then begin
  if not Class1 and not Class2 then begin
    WriteLn('Not a class 1 or class 2 modem');
    Halt;
  end;
end else begin
  WriteLn('Failed to identify modem, status: ', AsyncStatus);
  Halt;
end;

AddFaxEntry(Sender, '260-7151', 'EXCVT.APF', '');
FaxTransmitC12(Sender);
WriteLn('M`J' Results of fax send: ', AsyncStatus);

DoneC12SendFax(Sender);
DonePort(ComPort);
end.

```

As you can see, this program is almost identical to the OOP version. The only difference is that the port and send fax records are declared using pointer variables, which are initialized by the calls to `InitPort` and `InitC12SendFax`. These pointers are then passed to the other routines and finally disposed by the calls to `DoneC12SendFax` and `DonePort`.

Background Fax Transfer

Async Professional's implementation of fax transfers provides the ability to transfer faxes in the "background." Here, background means that the fax functions can perform just a small step of the overall fax process and then return control to your program. Your program can then perform a short task before looping around to call the fax transfer routine again.

This allows you to write applications that multiplex printing and fax reception, for example, and even to write memory resident programs that receive faxes in the background while your PC is running another application.

The reason that background transfers can work effectively is that, during fax transfers, your program spends much of its time waiting. It waits for incoming data during receives and waits for the serial output buffer to drain during transmits.

It's easiest to see how background transfers work by comparing a short background receive program to the equivalent program using the standard foreground approach. First, here's the standard approach, in OOP format:

```

var
  Receiver : C12ReceiveFaxPtr;
begin
  ...
  New(Receiver, Init('TEST', ComPort));

```

```

    Receiver^.FaxReceive;
    Dispose(Receiver, Done);
end.

```

This program instantiates a C12ReceiveFax object and starts receiving faxes.

Using the background fax functions, this program would be changed as follows:

```

var
    Receiver : C12ReceiveFaxPtr;
begin
    ...
    New(Receiver, Init('TEST', ComPort));
    Receiver^.PrepareFaxReceivePart; {<----- new}
    repeat {<----- new}
    until Receiver^.FaxReceivePart = faxFinished; {<----- new}
    Dispose(Receiver, Done);
end.

```

The call to FaxReceive is replaced with three new statements. PrepareFaxReceivePart initializes the object in preparation for the first call to FaxReceivePart. Each call to FaxReceivePart then performs a small portion of the fax receive process. FaxReceivePart is called repeatedly until it returns a function result of faxFinished, indicating that the receive session is done.

Called in a tight repeat loop like this, FaxReceivePart often has nothing to do; it's usually waiting for more data to arrive. Obviously, the place to perform other tasks is within the repeat loop. Your program can do almost anything here as long as each call doesn't take very long. Spending too long in any one call will probably cause FaxReceivePart to lose data, resulting in a poor quality fax. Using a large serial input buffer, say 16K bytes, can minimize this problem. Even so, the buffer will eventually overflow unless you call FaxReceivePart again. There is no way to use flow control to tell the remote faxmodem to temporarily stop sending data.

Notice that FaxReceivePart is a function that returns the enumerated FaxStateType. FaxStateType has four possible values:

- faxReady - There is work pending. Your program should call FaxReceivePart or FaxTransmitPart again immediately or the transfer speed will be reduced.
- faxCritical - FaxReceivePart or FaxTransmitPart is in a time-critical portion of the transfer. Your program must call FaxReceivePart immediately or risk failing the transfer.
- faxWaiting - FaxReceivePart or FaxTransmitPart is waiting for data to come in or go out. This is the best time for your program to perform other brief tasks.
- faxFinished - The fax receive session has completed either successfully or with an error. FaxReceivePart and FaxTransmitPart should not be called again until a new transfer is desired.

FaxTransmitPart and FaxReceivePart work with Class 1 and Class 2 faxmodems as well as CAS faxmodems. With CAS faxmodems only the states faxWaiting and faxFinished are returned. Because the CAS driver is controlling the faxmodem, there are no faxReady or faxCritical states.

See the FAXSRVR/FAXSRVRO source code for a practical example of multiplexing print jobs with fax reception.

Faxing From a TSR

A related feature of the Async Professional fax architecture is the ability to send and receive faxes in the background from a memory resident program. This doesn't mean just a TSR that pops up and

sends or receives a fax while popped up. Instead it means a TSR that transfers the fax in the background while your machine is running another program.

This feature is most useful for receiving faxes. For example, you can load a fax receive TSR (such as the supplied demo program RFAX or RFAXO) and then run other programs in the foreground as usual. The TSR is capable of detecting and answering an incoming call and receiving the fax without significantly disturbing your foreground program. You may notice that the foreground program pauses for a second or two at various times, perhaps for as long as 5 seconds when using Class 1 modems. Some of these pauses are due to file I/O, others occur when the faxmodem is at a time-critical portion of the transfer.

Async Professional doesn't provide any of the TSR or ISR routines necessary for creating a memory resident program. The following discussion assumes that you are using TSR routines from Object Professional, TSRs Made Easy, or Turbo Professional. However, any TSR routines with similar capabilities should work just as well.

A TSR fax program requires a clock interrupt handler and at least one popup entry point from which to call FaxTransmitPart or FaxReceivePart (henceforth referred to as FaxXxxPart). The clock interrupt handler is the mechanism that allows your TSR to work in the background. Your clock handler is called for every clock interrupt, about 18.2 times per second. That's too high a frequency to use when calling FaxXxxPart, since it will often have nothing to do. Instead, you'll want to increment a counter each time your clock interrupt handler is called and call FaxXxxPart at a lower frequency, say once per second. Be careful not to use too low a frequency or you may lose data or even fail the transfer.

Because the fax routines use DOS services, you cannot call them directly from your clock interrupt handler. Instead, you must schedule a popup procedure using the SetPopTicker routine. SetPopTicker calls your popup procedure as soon as DOS services are safely available. Your popup procedure can then call FaxXxxPart, display status information, and do just about anything else necessary to your application.

Of course, the foreground program is suspended while your popup procedure is running, so it should generally be as brief as possible. The exception to this is when FaxXxxPart returns faxCritical. In this case you must call FaxXxxPart again immediately and continue to call it until it returns a value other than faxCritical. Class 2 faxmodems return faxCritical for short periods, under 2 seconds, at several points in the transfer. Class 1 modems require slightly longer critical periods, 5 seconds or more, at those same points.

See the RFAX/RFAXO source code for examples.

Cover Pages

Fax transmissions often include a cover page, which provides basic information about the fax, usually consisting of the sender's name, the recipient's name, the total number of pages, and perhaps the date and time. Async Professional provides several options for sending this information. First, you can avoid a cover page altogether and put the same information on a fax page header. See the SetHeaderText routine (APFAX12/OOFAX12) for details about this method.

To send a separate cover page, you have three options. First, you can build the cover page right into the main APF file for the document. This option applies to text documents only. You just store your cover page text at the beginning of the document, insert a form feed character, and continue with the document itself. Then use the APFAXCVT/OOFAXCVT routines to generate an APF file.

Second, you can create a separate APF file that contains your cover page. Take this approach if the cover page contains special graphics or text that doesn't change between fax transmissions. Just

create the APF file beforehand and pass its name (including the 'APF' extension) as the Cover parameter to AddFaxEntry. FaxTransmit automatically transmits the Cover file before transmitting the body of the fax. The APF file used for the cover can contain one or more pages.

The third and most flexible approach is to put the cover page in a separate text file. If the cover file name you specify does *not* have an extension of 'APF' it is assumed to be text file. FaxTransmit automatically converts this to fax format using the built-in standard font and transmits it to the remote machine.

What makes this approach more flexible is the fact that the text file can use replacement tags, just the same as a fax page header does. As each line of the cover file is converted, the replacement tags are replaced with appropriate text. A line in the cover file can consist of any mix of tags and normal text (be careful that your normal text doesn't happen to contain tags, though). Blank lines and spaces can be used to format the cover page.

Here is a list of the available tags:

- \$D - today's date in MM/DD/YY format, always 8 characters
- \$I - station ID
- \$N - total number of pages, variable size
- \$P - current page number, variable size
- \$R - recipient's name as set by SetRecipientName, variable size
- \$F - sender's name as set by SetSenderName, variable size
- \$S - title as set by SetTitle, variable size
- \$T - current time in HH:MMpm format, always 7 characters

Note that some of the tags vary in length. For example, \$P would be replaced by '1' for the first page and '10' for the tenth page.

By taking advantage of replacement tags you can create a single cover page file that is used for every fax transfer. Below is a sample cover file and sample code that sets the replacement tags:

COVER.TXT:

Tots Learning Center
Colorado Springs, Colorado

DATE : \$D
TIME : \$T
FROM : \$F
TO : \$R

(form feed)

Program:

```
Sender^.SetSenderName('Beth');  
Sender^.SetRecipientName('Sherry');  
Sender^.AddFaxEntry('260-7151', 'FILE1.APF', 'COVER.TXT');  
Sender^.FaxTransmit;  
Sender^.ClearFaxEntries;  
Sender^.SetRecipientName('Mary');  
Sender^.AddFaxEntry('260-7151', 'FILE2.APF', 'COVER.TXT');
```

The file FILE1.APF is sent to 260-7151 using the cover page file COVER.TXT. The date and time tags are replaced with the current date and time. The sender's name is set to 'Beth' and the

recipient's name is set to 'Sherry'. The file FILE2.APF is also sent to the same number but this time the recipient's name is set to 'Mary'.

Note that the second cover page option is not available when faxes are sent via CAS services, because APCASFAX/OOCASFAX rely on the CAS driver to convert the cover page to a fax file.

The User Abort Hook

There will be times when you need to cancel a fax transfer while it is still in progress. This may be necessary if something goes wrong at the remote fax machine or if the user simply decides not to continue the transfer. The abort hook gives your program an opportunity to inform the fax routines that you want to prematurely stop the transfer.

Instead of defining a separate abort hook, the fax routines use the abort hook of the com port that is being used for the transfer. If your program already has installed a com port abort function, nothing extra is needed to make it work with the fax transfer. If you don't already have an abort function, you will almost certainly want to install one to support fax transfers.

The typical abort function checks the keyboard for characters such as <Esc> or <CtrlX> that the user enters to signal cancellation of the transfer. Because all fax transfers are made through modem connections, you may also want to check the modem's DCD (Data Carrier Detect) signal within the abort function. In this way, you can abort the transfer quickly if the connection is broken, without waiting for the timeouts built into the fax transfer routines.

See "User Abort Function" in §4.B for more information and examples.

The User Error Hook

After you call FaxTransmit or FaxReceive, you should use the AsyncStatus variable to check the final status of the transfer. Appendix B provides a complete list of all the error codes that can affect a fax transfer. Following is an annotated list of these errors:

ecFaxVoiceCall -	One end of the connection is a voice call, not a fax call.
ecFaxDataCall -	One end of the connection is a data call, not a fax call.
ecFaxBusy -	The called fax device is busy. FaxTransmit automatically retries each call for the number of times passed to the SetConnectAttempts routine. If the line is still busy, FaxTransmit terminates with the ecFaxBusy error code.
ecFaxNoFontFile -	The external font file APFAX.FNT could not be found. This file is used for converting header lines and cover pages to fax format.
ecFaxNoCASManager -	You instantiated a CASFax object but the CAS modem manager driver is not installed.
ecFaxInitError -	An error occurred in the initialization process. Usually means that the modem returned 'ERROR' at some point, which in turn usually means that the local modem is not a Class 1 or Class 2 faxmodem.
ecFaxTrainError -	The training procedure used to select a modulation rate for a Class 1 faxmodem did not succeed for any of the possible rates.
ecFaxSessionError -	An unspecified error occurred while sending or receiving the fax image. For Class 2 modems you can get a more exact error code by calling GetHangupResult.

Unfortunately these codes are not usually very helpful, since they describe activities controlled by the modem itself, over which your software has no control.

- ecFaxNoConnect - An incoming call was not successfully answered. Usually means that the modems could not agree on a modulation rate.
- ecFaxPageError - The remote fax did not report the results of the last page reception.

You can also install a user error handler which is called by the fax routines whenever an error occurs. Since some of the errors detected during a transfer are not fatal to the transfer, your error handler may obtain more information than a check of AsyncStatus would provide later.

As with the abort hook, the fax routines use the error hook of the associated com port rather than defining a different hook. See "User Error Procedure" in §4.B for more information about installing a com port error routine.

In a com port error routine used during a fax transfer, you may want to ignore line errors that would be taken more seriously in other situations. ecOverrunError, ecParityError, and ecFramingError indicate that one or more bytes of the fax image are corrupted. The fax image can easily survive a reasonable number of these errors. When a more serious error is reported to your error routine, the fax routines themselves will abort the transfer shortly thereafter, returning the final error in AsyncStatus.

The Fax Status Hook

When you call FaxTransmit or FaxReceive, control will not return to your program until one or more faxes have been sent or received, or a fatal error occurs. This can take anywhere from a few seconds to several hours depending on the number and size of the faxes being transferred.

The fax status hook provides a mechanism for your program to report the status of the transfer while FaxTransmit and FaxReceive are in progress. Your status procedure is called after every major event during the transfer (getting the remote's station ID, hanging up, etc.) and is also called on a regular basis if no other event has occurred. It is called no less frequently than once per second, or the amount of ticks specified by the typed constant DefStatusTimeout (default 18). It is also called whenever DefStatusBytes (default 10000) bytes of image data are received.

A fax status routine must be a global, far procedure of the following form:

OOP:

```
{ $F+ }
procedure MyFaxStatus(FP : AbstractFaxPtr; Starting, Ending : Boolean);
begin
    ...
end;
```

non-OOP:

```
{ $F+ }
procedure MyFaxStatus(FP : FaxRecPtr; Starting, Ending : Boolean);
begin
    ...
end;
```

Activate your status procedure by calling SetFaxStatusProc. If you are using the OOP fax units, you can override the virtual method FaxStatus instead.

The FP parameter of the fax status routine points to the fax transfer object or record. This pointer provides access to methods, functions, and data fields that may help to indicate the status of the transfer.

The Starting and Ending parameters are useful when you need to build a popup window to display your status information. Starting is True only on the first call to the status routine. It's your signal to allocate and display an initial window and to handle any other startup issues. Similarly, Ending is True only on the last call to the status routine. Here you would erase and deallocate the status window.

The Starting and Ending parameters are particularly simple to use when you are sending faxes. The status routine is called with Starting set to True almost immediately after entering FaxTransmit. After all faxes are sent, and just before exiting FaxTransmit, the status routine is called again with Ending set to True.

The use of Starting and Ending is more complicated when you are receiving faxes. The problem is that you usually don't know that a fax is arriving when you call FaxReceive. Instead, you are awaiting inbound faxes and you probably don't want to display a fax status window until a fax transfer is actually in progress. To simplify this, FaxTransfer doesn't call your status routine until it detects a ringing phone. At that point it does so with Starting set to True. It then calls the status routine on a regular basis until the fax reception is completed or an error occurs. Then it calls your status routine with Ending set to True. It does not call your status routine again until another ring is detected.

Note that you must force a return from FaxReceive. You can install a user abort hook, or you can set the "one fax" option by calling SetOneFax, which causes FaxReceive to return after receiving one fax.

If you are receiving faxes and you want a status window visible at all times, even when a transfer is not in progress, simply display the window before calling FaxReceive and erase it when FaxReceive returns. Remember, however, that the status procedure is not called regularly during the time that no fax transfer is in progress.

One of the most important pieces of information available to the fax status routine is stored in an internal variable of the APABSFAX/OOABSFAX units. This Word variable stores a code that indicates the current state of the fax transfer. The value can be obtained by calling the GetFaxProgress function or method of APABSFAX or OOABSFAX, respectively.

The following table shows the possible values of GetFaxProgress while faxes are being sent.

fpInitModem -	The faxmodem is being initialized for fax use. For Class 1 modems, this is nearly instantaneous; for Class 2 modems, the initialization takes a couple of seconds.
fpDialing -	The faxmodem is dialing the number and waiting for a response from the remote fax device.
fpSessionParams -	The local and remote fax devices completed negotiation of session parameters. The status procedure can now call GetSessionParams to get the negotiated bit rate, fax resolution, and error correction mode.
fpGotRemoteID -	A fax connection was established and the remote fax has reported its station ID. The status procedure can now call GetRemoteID to get the ID string.

fpSendPage -	Page data is currently being transmitted. The status procedure can call GetPageInfo to find out which page is being transmitted and how many bytes have been transferred so far.
fpSendPageStatus -	All data for the current page has been transmitted. The remote fax device is being told whether there are more pages to follow.
fpPageError -	The remote fax device did not successfully receive the page data and it should be sent again. FaxTransmit resends the page for a limited number of Retries. See SetMaxRetries.
fpPageOK -	The remote fax device received the page data successfully.

The following table shows GetFaxProgress values while faxes are being received.

fpWaiting -	The faxmodem is waiting for incoming fax calls.
fpAnswer -	The faxmodem is answering an incoming call.
fpIncoming -	The incoming call has been validated as a fax call.
fpSessionParams -	The local and remote fax devices completed negotiation of session parameters. The status procedure can now call GetSessionParams to get the negotiated bit rate, fax resolution, and error correction mode.
fpGotRemoteID -	A fax connection was established and the remote fax has reported its station ID. The status procedure can now call GetRemoteID to get the ID string.
fpGetPage -	Page data is currently being received. The status procedure can call GetPageInfo to find out which page is being received and how many bytes have been received so far.
fpGetPageResult -	The faxmodem just reported whether it received a page successfully. The status procedure can call GetLastPageStatus to find out the result.
fpCheckMorePages -	The faxmodem is waiting for the remote fax device to indicate whether it has more pages to send.
fpGetHangup -	The faxmodem is waiting for a response to a disconnect or hangup command.
fpGotHangup -	The faxmodem has disconnected or hung up.

Note that the fax status routine is also called at least one time when a fatal error occurs during a fax transfer. The status routine should check the value of AsyncStatus before acting on the value of GetFaxProgress. If AsyncStatus is not equal to ecOK, the GetFaxProgress value may be invalid or meaningless. See Appendix B for a list of AsyncStatus error codes.

The Fax Logging Hook

It's often important to have a log of all incoming and outgoing activity on a particular fax machine. You may use this for billing purposes, to recover the station ID or image of a fax whose printout was lost, or to determine which faxes were not successfully sent during an automated transfer.

The fax logging hook provides a mechanism for your program to keep such a log. Although the logging hook is similar to the status hook, the logging hook is not called so frequently. The status hook is called for at least 20 different send and receive states, and it is also called once per second during transfer of page data. The logging hook is called only once at the beginning and end of each transfer.

A fax logging routine must be a global, far procedure of the following form:

OOP:

```
{F+}
procedure MyFaxLog(FP : AbstractFaxPtr; Number : String;
                  FName : PathStr; Log : TLogFaxCode);
begin
  ...
end;
```

non-OOP:

```
{F+}
procedure MyFaxLog(FP : FaxRecPtr; Number : String;
                  FName : PathStr; Log : TLogFaxCode);
begin
  ...
end;
```

Activate your logging procedure by calling SetFaxLogProc. If you are using the OOP fax units, you can override the virtual method LogFax instead.

The FP parameter of the fax log routine points to the fax transfer object or record. This pointer provides access to methods, functions, and data fields that may help to indicate the status of the transfer.

When the program is sending a fax, the Number parameter is the phone number being called. When the program is receiving a fax, Number is the station ID of the remote faxmodem.

Similarly, FName is the name of the fax file being sent or received.

The Log parameter is of an enumerated type that indicates the condition at the time the logging routine is called. TLogFaxCode defines the following values:

lfaxTransmitStart -	A fax transmit is starting.
lfaxTransmitOk -	The fax was transmitted successfully.
lfaxTransmitFail -	The fax was not transmitted successfully.
lfaxReceiveStart -	A fax receive is starting.
lfaxReceiveOk -	The fax was received successfully.
lfaxReceiveSkip -	The incoming fax was rejected.
lfaxReceiveFail -	The fax was not received successfully.

For the particular case of lfaxReceiveSkip, the Number parameter contains the remote station ID, but the FName parameter is an empty string. The Accept Fax hook has already rejected the fax for one reason or another, so no file name is assigned to it.

Here's an example of a logging routine, extracted from the SIMPSNDO demo program.

```
{F+}
procedure MyFaxLog(FP : AbstractFaxPtr; Number : String;
                  FName : PathStr; Log : TLogFaxCode);
var
  FLog : Text;
```

```

begin
  Assign(FLog, 'SIMPSNDO.HIS');
  Append(FLog);
  if IOResult = 2 then
    Rewrite(FLog);
  if IOResult <> 0 then
    Exit;
  case Log of
    lfaxTransmitStart :
      WriteLn(FLog, TodayString, ' ', NowString,
        ' -- start fax transmit: ', FName,
        ' to ', Number);
    lfaxTransmitOk :
      WriteLn(FLog, TodayString, ' ', NowString,
        ' -- end fax transmit:', FName, 'M`J');
    lfaxTransmitFail :
      WriteLn(FLog, TodayString, ' ', NowString,
        ' -- fail fax transmit(', AsyncStatus,
        '): ', FName, 'M`J');
  end;
  Close(FLog);
  if IOResult <> 0 then ;
end;

```

This example simply appends the logging results to a text file named SIMPSNDO.HIS. The date, time, file name, and phone number are stored for each relevant fax sending event.

Note that a logging routine isn't limited to keeping a text log as shown above. It could also be used to print a received fax, archive it, or add it to a database.

See the demonstration programs SIMPSND/SIMPSNDO and SIMPRCV/SIMPRCVO for additional examples of logging procedures. The use of these demos is described in §10.G.

The Next Fax Hook

One of the advantages of computer-controlled faxing is that you can easily send one fax to many recipients, or multiple faxes to one recipient.

The next fax hook is used by FaxTransmit to provide an arbitrary degree of flexibility for selecting the files and phone numbers used during automated transmissions. When you call TransmitFax, it calls the next fax function to obtain each phone number and fax file name to transmit. TransmitFax does not return until this function indicates that there are no more faxes to send.

You probably don't need to write a next fax function of your own, since Async Professional installs a default function that is usually adequate. The default function, called NextFaxList, and another supplied function, AddFaxEntry, work together to manage a linked list of pending faxes. AddFaxEntry adds another node to this list. NextFaxList returns each successive node in the list.

Assuming that you've already initialized a fax send object (Sender), here's some code that shows how to use AddFaxEntry:

```

Sender.AddFaxEntry('260-7151', 'FAX1.APF', 'COVER.TXT');
Sender.AddFaxEntry('555-1212', 'FAX2.APF', '');
Sender.FaxTransmit;
Sender.ClearFaxEntries;

```

Each call to AddFaxEntry adds another fax to the queue. It specifies the phone number to call, the fax file to send, and an optional text file that is used as a cover page. The same or different values can be used for each parameter to each call. FaxTransmit sends the entries in the order they were

added, until all are sent or an error occurs. ClearFaxEntries clears the linked list, which is *not* cleared automatically by FaxTransmit.

You can probably imagine a number of situations where a different kind of next fax function would be desirable. For example, suppose you wanted to send the same ten fax files to 100 different sites. You'd need to call AddFaxEntry 1000 times to queue all of these transfers. Since each call to AddFaxEntry allocates a linked list node that uses about 200 bytes, you'd consume about 200K of heap space. A more efficient approach would be to initialize two arrays—one with the phone numbers and one with the fax files—and use two nested For loops to get all combinations of the numbers and files.

You might also want a different next fax function if your program allows the user to select faxes from an on-screen pick list. In that case, data structures native to the pick list could be used to supply the next fax name and phone number.

A next fax routine must be a global, far function of the following form:

OOP:

```
{ $F+ }
function MyNextFax(FP : AbstractFaxPtr; var Number : String;
                  var FName : PathStr; var Cover : PathStr) : Boolean;
begin
    ...
end;
```

non-OOP:

```
{ $F+ }
function MyNextFax(FP : FaxRecPtr; var Number : String;
                  var FName : PathStr; var Cover : PathStr) : Boolean;
begin
    ...
end;
```

Activate your next fax function by calling SetNextFaxFunc. If you are using the OOP fax units, you can override the virtual method NextFax instead.

The FP parameter of the function points to the fax send object or record. This pointer provides access to methods, functions, and data fields that may be useful to the next fax function.

The next fax function must return the next phone number (Number), fax file name (FName), and cover file name (Cover). FName must specify a fax file already converted into APF format. Cover specifies an optional text file that the send fax routines will convert to fax format on the fly. Return an empty string if you don't want to use a cover file.

If the next fax function has no more files to send, it should return False. In this case, it doesn't need to initialize Number, FName, and Cover. Otherwise, NextFax should return True.

The Fax Name Hook

When Async Professional's FaxReceive routine receives a fax, it must choose a filename for storing the incoming image. Unlike a file transfer protocol, the fax sender does not provide any name.

The fax name hook gives your application an opportunity to choose the file name. FaxReceive calls this hook when it needs to create a new fax data file.

Async Professional provides two built-in fax name functions. The first of these functions, FaxNameMD, is installed automatically. It returns filenames in the following format:

mmddnnnn.APF where mm is the current month
dd is the current day
nnnn is a sequence number starting at 0001

The first fax file received on January 1, 1994 would be named 01010001.APF. The second fax received that day would be 01010002.APF, and so on. If (on a very busy day!) more than 9999 faxes are received, FaxNameMD returns NONAME.APF with AsyncStatus set to ecTooManyFiles.

The other built-in function, FaxNameCount, returns names in the following format:

FAXnnnn.APF where nnnn is a sequence number starting at 0001

If more than 9999 faxes are received before the fax directory is cleaned out, FaxNameCount returns NONAME.APF with AsyncStatus set to ecTooManyFiles.

The file extension for both of the built-in functions is determined by the APMISC typed constant FaxFileExt, which defaults to 'APF'.

FaxNameMD and FaxNameCount select the sequence number by scanning the destination directory for existing files. They start with sequence number 0001 and scan sequentially for the first number without a matching file. Obviously, when more matching files exist, it takes longer to determine the sequence number. The time is probably not an issue as long as there are fewer than about 100 matching files.

If the delay is a problem, or if you'd like to use a different algorithm for naming the fax files, you can write your own fax name function. The fax name routine must be a global, far function of the following form:

OOP:

```
{ $F+ }  
function MyFaxName(FP : AbstractFaxPtr) : PathStr;  
begin  
  ...  
end;
```

non-OOP:

```
{ $F+ }  
function MyFaxName(FP : FaxRecPtr) : PathStr;  
begin  
  ...  
end;
```

Activate your fax name function by calling SetFaxNameFunc. For example, to activate the built-in function FaxNameCount in a non-OOP program, you'd use the following call:

```
SetFaxNameFunc(FP, FaxNameCount);
```

If you are using the OOP fax units, you can override the virtual method FaxName instead.

The FP parameter of the function points to the fax send object or record. This pointer provides access to methods, functions, and data fields that may be useful to the fax name function.

The fax name function must return the desired file name for the incoming fax. The name may include a drive and directory if desired, and should generally use the file extension FaxFileExt (default 'APF') for compatibility with the fax unpacking tools of Async Professional. If the specified file already exists, FaxReceive will overwrite it without any warning.

Here's an OOP example of a fax name function that does not scan the output directory to obtain a sequence number.

```
const
  LastNumber : Word = 0;

function MyFaxName(FP : AbstractFaxPtr) : PathStr;
var
  Numbers : String[4];
begin
  if LastNumber < 10000 then begin
    Inc(LastNumber);
    Str(LastNumber, Numbers);
    while Length(Numbers) < 4 do
      Numbers := '0'+Numbers;
    MyFaxName := 'FAX'+Numbers+'.'+FaxFileExt;
  end else begin
    AsyncStatus := ecTooManyFiles;
    MyFaxName := 'NONAME.APF';
  end;
end;
```

This function uses a typed constant named `LastNumber` that retains the last sequence number used since the program started. You might want to maintain the value of `LastNumber` in a program configuration file, or scan the fax directory once when your program starts up, to determine a good initial value for `LastNumber`. If you use the example as written, it will use sequence number 0001 every time the program starts up, overwriting any fax files that remain from a previous run of the program.

The Accept Fax Hook

The accept fax hook allows your program to determine whether to receive any particular incoming fax. `FaxReceive` calls the accept function as soon as it has established a fax session and determined the station ID of the sender.

There aren't many good reasons to refuse an incoming fax, but you may think of some that are important to your application. For example, in an automated fax receive system you may know that not enough disk space remains to hold even a small incoming fax. In this case, you'd save the sender's money by refusing the fax instead of accepting part of it and failing later.

Another possibility would be a system for rejecting junk faxes automatically. You could maintain a list of known-good station IDs whose faxes you will always accept, or a list of known-bad ones whose faxes you will always reject.

If you do not install an accept fax hook, `FaxReceive` accepts all incoming faxes. The accept fax hook cannot be used with CAS fax receives because the CAS driver is responsible for receiving the fax.

An accept fax routine must be a global, far function of the following form:

OOP:

```
{ $F+ }
function MyAcceptFax(FP : AbstractFaxPtr; RemoteName : Str20) : Boolean;
begin
  ...
end;
```

non-OOP:

```
($F+)  
function MyAcceptFax(FP : FaxRecPtr; RemoteName : Str20) : Boolean;  
begin  
    ...  
end;
```

Activate your accept fax function by calling SetAcceptFaxFunc. If you are using the OOP fax units, you can override the virtual method AcceptFax instead.

The FP parameter of the function points to the fax send object or record. This pointer provides access to methods, functions, and data fields that may be useful to the accept fax function.

The RemoteName parameter holds the station ID of the sender.

The accept fax function returns True to accept the fax, or False to reject it.

Here's a simple OOP example of an accept fax function:

```
function MyAcceptFax(FP : AbstractFaxPtr; RemoteName : Str20) : Boolean;  
begin  
    MyAcceptFax := (RemoteName = 'FIRST STREET OFFICE');  
end;
```

This function rejects all faxes but those coming from the 'FIRST STREET OFFICE' station ID.

If your accept fax function rejects a fax, FaxReceive informs the sender and then returns control to your application with AsyncStatus set to ecFileRejected.

Shared Declarations

Constants

```
afAbortNoConnect      = $0001;  
afCASWaitTillDone     = $0002;  
afExitOnError         = $0004;  
afCASSubmitUseControl = $0008;
```

Fax send/receive options. If afAbortNoConnect is set, FaxTransmit returns to the calling application if it is unable to connect to a particular fax number after the default number of retries, even if additional faxes are already queued to be sent. If afAbortNoConnect is not set, FaxTransmit moves on to the next entry returned by the next fax hook.

If afCASWaitTillDone is set, as it is by default, the CAS version of FaxTransmit remains in control until the CAS driver indicates that the fax has been fully transmitted, and the CAS version of FaxReceive remains in control until an abort hook or an error forces it to return. This option makes the APFAXCAS/OOFAXCAS versions of FaxTransmit and FaxReceive act more like the APFAX12/OOFAX12 versions, especially in terms of how the fax user hooks are called. If afCASWaitTillDone is not set, FaxTransmit returns immediately after all fax requests are submitted to the CAS driver, and FaxReceive returns as soon as any already-queued faxes are transferred to DOS files.

If an error occurs within FaxTransmit or FaxReceive when the afExitOnError option is enabled, control immediately returns to your application. If an error occurs when the afExitOnError option is disabled, as it is by default, FaxTransmit continues on with the next fax in the transmit queue, and FaxReceive resets the modem and starts listening for a new incoming fax.

If afCASSubmitUseControl is enabled, SubmitSingleFile does *not* use the CAS \$15 function, submit single file. Instead, it formats and creates a control file and submits the control file using SubmitTask. This avoids a bug in at least one version of CASMODEM where function \$15 does not send the requested cover text. An added advantage of this option is that it uses the PCX logo

file set by SetLogoFile, which isn't used by the normal SubmitSingleFile function.
afCASSubmitUseControl is disabled by default.

```
BadFaxOptions = 0;
```

Fax send/receive options for internal use, which cannot be changed by calling afOptionsOn or afOptionsOff. There are currently no such options.

```
DefConnectAttempts : Word = 1;      {Connect attempts when dialing}
DefMaxRetries : Integer = 2;        {Send retries for a page with errors}
DefCmdTimeout : Integer = 546;      {Ticks to wait for a command response}
DefDialTimeout : Integer = 1092;    {Ticks to wait for a dial response}
DefTransTimeout : Integer = 1092;   {Ticks to wait for output buffer room}
DefStatusTimeout : Integer = 18;    {Ticks between status updates}
DefStatusBytes : Word = 10000;      {Image bytes between status updates}
MaxBadPercent : Word = 10;          {Percent bad bytes allowed during
                                     successful training}
```

Timing and retry parameters used during fax send/receive. These typed constants are copied into the fields of a fax record or object when it is initialized. Connect attempts can be modified later, by calling the SetConnectAttempts routine. Page retries can also be modified later, by calling the SetMaxRetries routine of APFAX12/OOFAX12. Generally speaking, these constants apply only to Class 1 and Class 2 fax transfers. CAS transfers are tuned by using the CAS driver software.

```
DefFaxOptions : Word = afCASWaitTillDone;
```

Default fax send/receive options.

```
DefInit = 'ATE0Q0V1X4S0=0';
```

This string is always sent to the Class 1 or Class 2 faxmodem when a send or receive is starting. It is sent immediately after the string specified by SetModemInit (APFAX12/OOFAX12) is sent.

Types

```
FaxStateType = (
    faxReady,      {State machine ready immediately}
    faxWaiting,    {State machine waiting}
    faxCritical,   {State machine in critical state}
    faxFinished); {State machine is finished}
```

Fax states returned by FaxTransmitPart and FaxReceivePart. faxReady indicates that more work can be done immediately—you should call FaxTransmitPart or FaxReceivePart soon, but not doing so will not damage the fax session. faxWaiting indicates the fax session is waiting for a response, for more data, or for a buffer to drain—your program can safely perform a brief task, perhaps up to several seconds long, before calling FaxTransmitPart or FaxReceivePart again. faxCritical means that the fax session is in a time critical area—you *must* call FaxTransmitPart or FaxReceivePart again immediately. faxFinished means that the fax session is finished.

```
TLogFaxCode = (
    lfaxTransmitStart, {A fax transmit session is starting}
    lfaxTransmitOk,    {The fax was transmitted successfully}
    lfaxTransmitFail,  {The fax was not transmitted successfully}
    lfaxReceiveStart,  {A fax receive session is starting}
    lfaxReceiveOk,     {The fax was received successfully}
    lfaxReceiveSkip,   {The incoming fax was rejected}
    lfaxReceiveFail); {The fax was not received successfully}
```

Codes passed to the fax logging procedure. See "The Fax Logging Hook" earlier in this section for more information.

APABSFAX Declarations

Types

```
AcceptFaxFunc = function(FP : FaxRecPtr; RemoteName : Str20) : Boolean;
```

A function of this type is used to accept or reject an incoming fax. See "The Accept Fax Hook" earlier in this section for more information.

```

FaxLogProc = procedure(FP : FaxRecPtr; Number : String; FName : PathStr;
    Log : TLogFaxCode);

```

A procedure of this type is used to log the start and end of each fax that is sent or received. See "The Fax Logging Hook" earlier in this section for more information.

```

FaxNameFunc = function(FP : FaxRecPtr) : PathStr;

```

A function of this type is used to return a file name for each incoming fax. See "The Fax Name Hook" earlier in this section for more information.

```

FaxRecPtr = ^FaxRec;

```

```

FaxRec = record

```

```

    ...

```

```

end;

```

A generic fax send or receive record. A pointer of type FaxRecPtr is initialized by InitXxxYyy (e.g., InitC12SendFax) and must then be passed to all non-OOP fax send/receive routines. You should not need to refer to any of the fields of this record.

```

FaxStatusProc = procedure(FP : FaxRecPtr; Starting, Ending : Boolean);

```

A procedure of this type is used to display fax send or receive status information. See "The Fax Status Hook" earlier in this section for more information.

```

NextFaxFunc = function(FP : FaxRecPtr; var Number : String;
    var FName : PathStr; var Cover : PathStr) : Boolean;

```

A function of this type is used to return the next fax phone number, image file name, and cover file name to FaxTransmit. See "The Next Fax Hook" earlier in this section for more information.

OOABSFAX Declarations

Types

```

AbstractFaxPtr = ^AbstractFax;

```

```

AbstractFax =
    object(Root)

```

```

    ...

```

```

end;

```

The base fax send/receive object, used as an ancestor to the Class 1/2 and CAS fax objects. A pointer of type AbstractFaxPtr is passed to the fax send/receive user hooks, where it may be typecast if necessary to more specific types.

```

AcceptFaxFunc = function(FP : AbstractFaxPtr; RemoteName : Str20) : Boolean;

```

A function of this type is used to accept or reject an incoming fax. See "The Accept Fax Hook" earlier in this section for more information.

```

FaxLogProc = procedure(FP : AbstractFaxPtr; Number : String; FName : PathStr;
    Log : TLogFaxCode);

```

A procedure of this type is used to log the start and end of each fax that is sent or received. See "The Fax Logging Hook" earlier in this section for more information.

```

FaxNameFunc = function(FP : AbstractFaxPtr) : PathStr;

```

A function of this type is used to return a file name for each incoming fax. See "The Fax Name Hook" earlier in this section for more information.

```

FaxStatusProc = procedure(AP : AbstractFaxPtr; Starting, Ending : Boolean);

```

A procedure of this type is used to display fax send or receive status information. See "The Fax Status Hook" earlier in this section for more information.

```

NextFaxFunc = function(AP : AbstractFaxPtr; var Number : String;
    var FName : PathStr; var Cover : PathStr) : Boolean;

```

A function of this type is used to return the next fax phone number, image file name, and cover file name to FaxTransmit. See "The Next Fax Hook" earlier in this section for more information.

Syntax

```
procedure AddFaxEntry(FP : FaxRecPtr; Number : String; FName : PathStr;  
                     Cover : PathStr);  
  
procedure AbstractFax.AddFaxEntry(Number : String; FName : PathStr;  
                                  Cover : PathStr);
```

Purpose

Add a fax entry to the built-in list.

Description

This routine adds the specified phone number, fax file name, and cover file name to the built-in linked list of fax entries. It works in conjunction with the default next fax hook, NextFaxList. FaxTransmit sends all the faxes on this list in the order they were added, returning only when it is finished or a fatal error occurs.

Number is the complete phone number of the receiving fax machine. It may include modem dial modifiers such as ',' or '@', but it must *not* include the 'ATDT' prefix. FName is the file name, including path if needed, of a fax image file in APF format.

Cover is the file name, including path if needed, of an optional text file that will be converted to fax format and sent before the main body of the fax. Specify an empty string for Cover if you don't want a cover sheet. When FaxTransmit converts the Cover file to a fax file, it uses the font loaded by LoadFont or LoadHPFont (APFAXCVT/OOFAXCVT). However, it doubles the normal height of the characters. Hence, you should generally limit the Cover file size to no more than 30 text lines.

Each call to AddFaxEntry allocates about 200 bytes of heap space. Check AsyncStatus after the call if there's a possibility that you will run out of memory.

The list of nodes created by calls to AddFaxEntry is not disposed by FaxTransfer. To clear the list, call ClearFaxEntries or dispose of the fax send object or record.

Example

```
AddFaxEntry(FP, '260-7151', 'FAX1.APF', 'COVER.TXT');  
AddFaxEntry(FP, '555-1212', 'FAX2.APF', '');  
FaxTransmitC12(FP);  
ClearFaxEntries(FP);
```

Adds two entries to the list of faxes to be transmitted, transmits the faxes, then clears the list. When using objects, the FP parameters and the 'C12' suffix would be omitted.

See Also

ClearFaxEntries

afOptionsAreOn / afOptionsOff / afOptionsOn

Syntax

```
function afOptionsAreOn(FP : FaxRecPtr; OptionFlags : Word) : Boolean;  
function AbstractFax.afOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose

Return True if all specified fax options are on.

Description

This function returns True if the specified fax options are currently selected.

Example

```
if afOptionsAreOn(F, afAbortNoConnect) then  
  afOptionsOff(F, afAbortNoConnect)  
else  
  afOptionsOn(F, afAbortNoConnect);
```

Toggle the state of the afAbortNoConnect option. When using objects, the F parameters would be omitted.

Syntax

```
procedure afOptionsOff(FP : FaxRecPtr; OptionFlags : Word);  
procedure AbstractFax.afOptionsOff(OptionFlags : Word);
```

Purpose

Deactivate multiple fax options.

Description

This procedure deactivates the specified conversion options, excluding those designated as BadFaxOptions.

Example

See the example for afOptionsAreOn.

Syntax

```
procedure afOptionsOn(FP : FaxRecPtr; OptionFlags : Word);  
procedure AbstractFax.afOptionsOn(OptionFlags : Word);
```

Purpose

Activate multiple fax options.

Description

This procedure activates the specified conversion options, excluding those designated as BadFaxOptions.

Example

See the example for afOptionsAreOn.

Syntax

```
procedure ClearFaxEntries(FP : FaxRecPtr);  
procedure AbstractFax.ClearFaxEntries;
```

Purpose

Remove all fax entries from the built-in list.

Description

This routine removes all entries from the list of faxes created by AddFaxEntry. The memory used by all entries is released and the list count is set to zero.

Note that the list is not cleared automatically by FaxTransmit. It is cleared only when you call ClearFaxEntries or when you dispose of the fax send object or record. The following OOP code fragment is probably in error:

```
AddFaxEntry('260-7151', 'TEST.APF', '');  
FaxTransmit;  
AddFaxEntry('260-7151', 'NEXT.APF', '');  
FaxTransmit;
```

The second call to FaxTransmit transmits *both* TEST.APF and NEXT.APF because the list was never cleared after the first call to FaxTransmit.

Example

See the example for AddFaxEntry.

Syntax

```
destructor AbstractFax.Done; virtual;
```

Purpose

Dispose of a fax object.

Description

This destructor is used internally by all objects derived from AbstractFax. Its main purpose is to release any memory used by the built-in fax entry list. It should be called only by the destructors of objects derived from AbstractFax.

GetFaxName

Syntax

```
function GetFaxName(FP : FaxRecPtr) : PathStr;  
function AbstractFax.GetFaxName : PathStr;
```

Purpose

Return name of current fax if known.

Description

This function is intended to be used in status routines. It returns the name of the fax file currently being sent or received.

When you are transmitting a fax, this function returns a file name if it is called any time after GetFaxProgress returns fpInitModem. Before then, it returns an empty string.

When you are receiving a fax, this function returns a file name if it is called any time after GetFaxProgress returns fpGetPageData. Before then, it returns an empty string.

Example

```
case GetFaxProgress of  
  ...  
    fpInitModem:  
      WriteLn(GetFaxName(F));  
  ...  
end;
```

Part of a fax status procedure for a transmitting application. When using objects, the F parameter would be omitted.

See Also

GetPageInfo (APFAX12)
C12AbstractFax.GetPageInfo
GetSessionParams (APFAX12)

C12AbstractFax.GetSessionParams
GetRemoteID (APFAX12)
C12AbstractFax.GetRemoteID

Syntax

```
function GetFaxProgress(FP : FaxRecPtr) : Word;  
function AbstractFax.GetFaxProgress : Word;
```

Purpose

Return the current fax progress value.

Description

This function is intended for use in status routines. It returns an fpXxx value representing the current state of the fax transfer.

Calling GetFaxProgress is the first thing a typical status procedure should do. The status procedure can then display an appropriate message or, depending on the fpXxx value returned, call another APABSFAX/OOABSFAX information routine.

See "The Fax Status Hook" for more information and a list of fpXxx values.

Example

```
var  
    Progress : Word;  
begin  
    Progress := GetFaxProgress(F);  
    case Progress of  
        fpInitModem :  
            WriteLn('Initializing fax modem');  
        fpDialing :  
            begin  
                WriteLn('fax file to send: ', GetFaxName(F));  
                WriteLn('Dialing/waiting for connection');  
            end;  
        ...  
        fpGotRemoteID :  
            WriteLn('Connected to ', GetRemoteID(F));  
        ...  
    end;  
end;
```

Part of a status routine that displays various messages based on the value returned by GetFaxProgress. Notice that other routines, GetFaxName and GetRemoteID, are called in response to certain fpXxx values. When using objects, the F parameters would be omitted.

Syntax

```
constructor AbstractFax.Init(ID : Str20);
```

Purpose

Initialize low-level fields in the AbstractFax object.

Description

This constructor initializes the data fields of the AbstractFax object. It allocates no heap space. It should be called only by the constructor for fax objects derived from AbstractFax.

See Also

Done

SetAcceptFaxFunc / SetConnectAttempts

Syntax

```
procedure SetAcceptFaxFunc(FP : FaxRecPtr; AFF : AcceptFaxFunc);  
procedure AbstractFax.SetAcceptFaxFunc(AFF : AcceptFaxFunc);
```

Purpose

Activate an accept fax function.

Description

This routine lets you specify a function that is called during fax reception to accept or reject an incoming fax. The accept fax function is called as soon as the remote station ID is known.

The function passed as AFF must be far and global, and must match the form of the AcceptFaxFunc type.

If you don't call SetAcceptFaxFunc, all incoming faxes will be accepted. To deactivate an accept fax function, call SetAcceptFaxFunc and pass nil as the AcceptFaxFunc.

See "The Accept Fax Hook" earlier in this section for more information and examples.

Example

```
SetAcceptFaxFunc(F, MyAcceptFax);
```

Activates MyAcceptFax as the accept fax function. When using objects, the F parameter would be omitted.

Syntax

```
procedure SetConnectAttempts(FP : FaxRecPtr; Attempts, DelayTicks : Word);  
procedure AbstractFax.SetConnectAttempts(Attempts, DelayTicks : Word);
```

Purpose

Set the connect retry parameters for a fax transmission.

Description

When FaxTransmit is sending a fax and the faxmodem's response to the dial attempt is 'BUSY' or 'NO ANSWER', FaxTransmit can automatically retry the call. If you don't call SetConnectAttempts, it attempts to connect DefConnectAttempts times. This typed constant defaults to one, so no retrying is performed.

Call SetConnectAttempts with Attempts set to any value other than one to activate automatic retry. If you pass zero for this parameter, FaxTransmit retries forever, until it either makes a connection or is cancelled by a user abort function. If you pass two for Attempts, SetConnectAttempts retries one time.

Before FaxTransmit dials again, it waits DelayTicks ticks.

This routine does not affect CAS fax transfers. The number of retries for CAS can be adjusted only by the CAS setup software supplied with the faxmodem.

Example

```
SetConnectAttempt(F, 10, 1092);
```

Sets the total number of connect attempts to 10 with a delay of 1092 ticks, or about one minute, between attempts. When using objects, the F parameter would be omitted.

Syntax

```
procedure SetFaxLogProc(FP : FaxRecPtr; FLP : FaxLogProc);  
procedure AbstractFax.SetFaxLogProc(FLP : FaxLogProc);
```

Purpose

Activate a fax logging routine.

Description

This routine allows you to specify a procedure that is called at the beginning and end of each fax transmission and reception. Within your routine you can update a log of fax events, print received faxes, and so on. This hook is particularly useful for unattended fax applications.

The procedure passed as FLP must be far and global, and must match the form of the FaxLogProc type.

If you don't call SetFaxLogProc, no logging will be performed. To deactivate a logging routine, call SetFaxLogProc and pass nil as the FaxLogProc.

See "The Fax Logging Hook" earlier in this section for more information and examples.

Example

```
SetFaxLogProc(F, MyFaxLog);
```

Activates MyFaxLog as the logging procedure. When using objects, the F parameter would be omitted.

Syntax

```
procedure SetFaxNameFunc(FP : FaxRecPtr; FNF : FaxNameFunc);  
procedure AbstractFax.SetFaxNameFunc(FNF : FaxNameFunc);
```

Purpose

Activate a fax naming function.

Description

This routine allows you to specify a function that is called when an incoming fax is detected and FaxReceive needs a file name where it will store the fax image.

The function passed as FNF must be far and global, and must match the form of the FaxNameFunc type.

If you don't call SetFaxNameFunc, FaxReceive uses a built-in fax name function called FaxNameMD. This function names incoming faxes using the form 'mmddnnnn.APF', where mm is the month number, dd is the day, and nnnn is a sequence number starting at one. You can also activate another built-in fax name function called FaxNameCount. It names faxes using the form 'FAXnnnn.APF', where nnnn is a sequence number.

See "The Fax Name Hook" earlier in this section for more information and examples.

Example

```
SetFaxNameFunc(F, MyFaxName);
```

Activates MyFaxName as the fax naming function. When using objects, the F parameter would be omitted.

SetFaxStatusProc / SetNextFaxFunc

Syntax

```
procedure SetFaxStatusProc(FP : FaxRecPtr; SP : FaxStatusProc);  
procedure AbstractFax.SetFaxStatusProc(SP : FaxStatusProc);
```

Purpose

Activate a fax status routine.

Description

This routine allows you to specify a function that is called regularly by FaxTransmit and FaxReceive to display fax status information.

The function passed as SP must be far and global, and must match the form of the FaxStatusProc type.

If you don't call SetFaxStatusProc, no status will be displayed. To deactivate a status function, call SetFaxStatusProc and pass nil as the FaxStatusProc.

See "The Fax Status Hook" earlier in this section for more information and examples.

Example

```
SetFaxStatusProc(F, MyFaxStatusProc);
```

Activates MyFaxStatusProc as the fax status routine. When using objects, the F parameter would be omitted.

See Also

SetFaxLogProc

Syntax

```
procedure SetNextFaxFunc(FP : FaxRecPtr; NFF : NextFaxFunc);  
procedure AbstractFax.SetNextFaxFunc(NFF : NextFaxFunc);
```

Purpose

Activate a next fax function.

Description

This routine allows you to specify a function that is called by FaxTransmit to obtain the next fax to send.

The function passed as NFF must be far and global, and must match the form of the NextFaxFunc type.

If you don't call SetNextFaxFunc, FaxTransmit uses a built-in next fax function called NextFaxList. This function returns the next fax from a linked list of pending faxes. You can add faxes to this list by calling AddFaxEntry.

See "The Next Fax Hook" earlier in this section for more information and examples.

Example

```
SetNextFaxFunc(F, MyNextFax);
```

Activates MyNextFax as the next fax function. When using objects, the F parameter would be omitted.

SetRecipientName / SetSenderName

Syntax

```
procedure SetRecipientName(FP : FaxRecPtr; NewName : String);  
procedure AbstractFax.SetRecipientName(NewName : String);
```

Purpose

Add a name for the token \$R for cover files.

Description

The NewName string replaces the \$R token in the cover page text file when a fax is transmitted. The \$R token can also be used in header lines, but the cover page is the more likely place to provide this information.

See "Cover Pages" earlier in this section for information and examples.

Example

```
SetRecipientName(F, 'Megan');  
SetSenderName(F, 'Stephanie');  
AddFaxEntry(F, Number, FileName, 'COVER.TXT');  
FaxTransmitC12(F);
```

If the cover page text file COVER.TXT contains \$R and \$F tokens they will be replaced with 'Megan' and 'Stephanie', respectively.

See Also

SetSenderName

SetTitle

Syntax

```
procedure SetSenderName(FP : FaxRecPtr; NewName : String);  
procedure AbstractFax.SetSenderName(NewName : String);
```

Purpose

Add a name for the token \$F for cover files.

Description

The NewName string replaces the \$F token in the cover page text file when a fax is transmitted. The \$F token can also be used in header lines, but the cover page is the more likely place to provide this information.

See "Cover Pages" earlier in this section for information and examples.

Example

See SetRecipientName for an example.

See Also

SetRecipientName

SetTitle

SetStationID / SetTitle

Syntax

```
procedure SetStationID(FP : FaxRecPtr; NewID : Str20);  
procedure AbstractFax.SetStationID(NewID : Str20);
```

Purpose

Set the faxmodem's station ID.

Description

A fax device can identify itself to another fax device with a 20 character name, called the station ID. The Class 1 and Class 2 specifications indicate that the station ID should contain just a phone number, therefore they limit it to just the digits 0 through 9 and space. However, the station ID is frequently used to store an alphabetic name. Most faxmodems support this convention by allowing upper and lower case letters as well as other special characters in the station ID. This can cause problems for some fax machines, though, since they cannot print these characters.

Async Professional does not filter the characters stored in the station ID. You may wish to limit the characters passed to SetStationID in order to make your software compatible with the broadest possible range of fax hardware.

The station ID, NewID, is used on both incoming and outgoing calls.

Note that a fax file stored in the Async Professional APF format also contains a station ID in the file header. This station ID is stored when a document is converted to APF format. To use this station ID when you transmit the fax, call GetFaxHeader (APFAXCVT/OOFAXCVT) to get the file's station ID, then call SetStationID (APABSFAX/OOABSFAX) to set the station ID for the next fax transmission.

Examples

```
SetStationID(F, '1 719 260 7151');
```

Specifies a station ID that conforms to the Class 1 and Class 2 specifications.

```
SetStationID(F, 'HOME OFFICE');
```

Specifies a station ID that doesn't conform to the Class 1 and Class 2 specifications, but works on most modern fax machines.

Syntax

```
procedure SetTitle(FP : FaxRecPtr; NewTitle : String);  
procedure AbstractFax.SetTitle(NewTitle : String);
```

Purpose

Specify a title for the transmitted fax header line.

Description

The title string is inserted into the fax header string that you have specified with SetHeaderText (APFAX12/OOFAX12). When the fax is transmitted, the title string replaces the '\$S' token in the header string.

Title and header strings are not applicable to faxes transmitted using a CAS driver.

Example

```
SetTitle(F, 'REPORT1');  
FaxTransmitC12(F);  
...  
SetTitle(F, 'REPORT2');  
FaxTransmitC12(F);
```

Uses a title of REPORT1 for the first fax transmission, then switches to REPORT2 for the second transmission.

See Also

SetHeaderText (APFAX12)

C12SendFax.SetHeaderText

E. Class 1/2 Fax Send/Receive: APFAX12/OOFAX12

APFAX12/OOFAX12 build onto APABSFAX/OOABSFAX to provide fax send and receive services for Class 1 and Class 2 faxmodems. Use APFAX12 or OOFAX12 when your application is controlling the faxmodem directly and not using a faxmodem manager such as CAS.

Class 1 and Class 2 faxmodems provide quite similar capabilities. Both can connect to any other Group 3 fax device, both use the same image transmission format, and both are capable of transferring data at the same speeds. The differences between the two are all in the area of how the PC terminal software interacts with the faxmodem.

When using a Class 1 faxmodem, the communication between the PC and the modem is at a fairly low level. They exchange information and commands through HDLC (High-level Data Link Control) packets instead of Hayes-type 'AT' sequences. The terminal software is also responsible for negotiating the faxmodem-to-faxmodem data transfer rate by sending and receiving training sequences.

When using a Class 2 faxmodem, the communication between the PC and the modem is based on an extended set of Hayes-type 'AT' commands and text responses. The faxmodem itself negotiates the faxmodem-to-faxmodem data transfer rate and reports the results back to the PC.

Because Class 1 and Class 2 faxmodems are used in such a similar fashion, APFAX12/OOFAX12 support them with a single set of routines and objects. In general, you don't need to be concerned with whether a Class 1 or Class 2 faxmodem is attached to the systems that your software supports. You call the same functions in either case.

By contrast, APFAX12/OOFAX12 separate the records/objects used for sending and receiving faxes. This allows Turbo Pascal's smart linker to more effectively reduce the size of programs that just send or just receive faxes. This separation also allows your program to use less heap space by allocating the send record/object only when it is needed, and disposing of it before receiving a fax.

In APFAX12, `InitC12SendFax` initializes a `FaxRecPtr` capable of sending faxes, and `DoneC12SendFax` disposes of it. `InitC12ReceiveFax` initializes a `FaxRecPtr` capable of receiving faxes, and `DoneC12ReceiveFax` disposes of it.

In OOFAX12, the `C12SendFax` object type sends Class 1 and Class 2 faxes. The `C12ReceiveFax` object receives them. To further improve the efficiency and maintainability of these objects, they are both derived from an intermediate object called `C12AbstractFax`, which implements capabilities that are shared between the two. You shouldn't instantiate an object of type `C12AbstractFax`, but some of its capabilities are documented here since they are inherited by the other two object types.

The Fax Transfer Process

This section describes the anatomy of a fax transfer in some detail. You don't need to know all of this to use the Async Professional fax units. However, it may be helpful for you to understand the detailed differences between Class 1 and Class 2, and to know some of the limitations of Group 3 faxmodems in general. The discussion also clarifies exactly when the various fax user hooks are called by `FaxTransmit` and `FaxReceive`.

The following diagram shows the steps followed by all Class 1 and Class 2 fax transfers.

Phase Name	Activity
Phase A	dial
Phase B	pre-message
Phase C	message
Phase D	post-message
Phase E	hangup

The "phase name" is the term used by the TIA/EIA specification for each activity. Phase A is associated with dialing the phone number in FaxTransmit or preparing the modem to receive a call in FaxReceive. The sender and receiver negotiate the parameters of the fax connection during Phase B. The actual fax image (the "message") is transferred during Phase C. During Phase D, the sender and receiver decide whether a page must be resent, or whether more pages follow. And during Phase E, the modem disconnects from the phone line in preparation for the next call.

All phases from A through E are processed in order unless an error occurs. Class 1 and Class 2 modems differ only in their processing of Phase B and Phase D. With Class 1 modems, the terminal software handles these two phases by transmitting and receiving HDLC packets. With Class 2 modems, the terminal sends an 'AT' sequence and waits for the modem to return a text response.

The following subtopics discuss each phase in even more detail.

Phase A - Dial

The first step in Phase A is to initialize the modem for sending or receiving faxes. FaxTransmit continues Phase A by calling the next fax hook to get the phone number and fax file name; if another fax is queued for sending, it dials the fax recipient's phone number.

Phase B - Pre-message

Phase B starts as soon as the call is answered. The major task of Phase B is for the modems to agree on a set of capabilities to use while transmitting this fax. The modems exchange their capabilities, then perform an iterative training procedure to find the fastest transmission parameters that are reliable.

During training, the sender starts at the fastest modulation rate that the receiver supports. It sends a known sequence of bytes, which the receiver attempts to read. If the receiver gets the right pattern, it informs the sender and the transfer moves on to Phase C. If the pattern is incorrect, the receiver informs the sender and the sender determines whether to retry at the same rate or to step down to the next lower modulation rate. The process is then repeated. Generally a reduction in transfer rate is required because the telephone line is too noisy. If no available modulation rate succeeds, the connection is terminated.

After a modulation rate is successfully defined, various user hooks are called. In FaxReceive, the accept fax hook is called to accept or reject this fax. If the fax is rejected, the logging hook is called to log the rejection. If the fax is accepted, the fax name hook is called to get a name for the received file, and the logging hook is called to log the start of the receive. In FaxTransmit, the fax logging hook is called to log the start of the transmit session.

Phase C - Message

During Phase C the sender's terminal software sends page image data to the faxmodem, which transmits it to the receiving faxmodem. The receiving faxmodem sends it on to the receiver's terminal. During this process, the sender's terminal must honor one-way software flow control,

because the transmitting faxmodem may send it an Xoff to temporarily stop the data flow. It sends an Xon when it wants to resume.

APFAX12/OOFAX12 enable and disable software flow control at the appropriate times within the Async Professional device layer. However, you must assure that software flow control is available to the device layer. This is automatic for all device layers except APUART, where you must select a UART configuration (in APDEFINE.INC) that supports software flow control. The default APDEFINE.INC UART configuration is "Standard," which does support software flow control.

The sender's terminal may need to send a particular modem setup string to the faxmodem so that it handles flow control correctly during the fax process. See SetModemInit in the reference section for additional details.

You may also want to enable hardware flow control in the sender's device layer, since some faxmodems use it in addition to software flow control. This step is optional, but marginally more efficient than software flow control.

While in Phase C the sender's terminal must never stop sending data to the faxmodem except when it is blocked by flow control. The faxmodem must send a constant stream of data to the receiver. Any break in the stream generates what is called a data underflow error. What happens after a data underflow depends on the faxmodem, but in most cases you won't be able to continue the fax session.

In order to avoid data underflow, you *must* select a port baud rate that is higher than (not just equal to) the fax bit per second transfer rate. In almost all cases you should select a port baud rate of 19200. There's no benefit to using higher baud rates and there is at least one modem we've tested that will work only at 19200.

The only time you wouldn't choose a port baud rate of 19200 is when your application needs to run on machines as slow as the original PC/XT. These machines have trouble sending or receiving large amounts of data at 19200 baud. In this case you should use a port speed of 9600 and force the fax data transfer rate to 4800 BPS or less. See SetModemFeatures in the reference section to see how.

Another possible cause of data underflow is the use of FaxTransmitPart to multiplex fax transmission with other tasks in your program. If FaxTransmitPart returns control to your application, you execute a non-fax task, and the serial output buffer drains before the task returns, data underflow will occur and the fax session will fail. The best way to avoid this is to use a relatively large UART output buffer, say 16384 bytes, and check that the buffer is relatively full before you execute a non-fax task.

During Phase C, the fax status hook is called regularly.

Phase D - Post-message

At the end of Phase C the transmitter sends an end-of-page sequence, which marks the start of Phase D. The receiver then tells the transmitter whether or not the page was received successfully. If it was not, the two modems can negotiate a retransfer. If it was, the transmitter tells the receiver whether or not any additional pages are coming.

If there are more pages, the process can loop back to either the middle of Phase B (to retrain the connection) or the beginning of Phase C (to keep the existing connection parameters). The receiving modem decides which approach to use. Async Professional's receive routines always loop to the beginning of Phase C; other software packages and fax machines may choose either approach.

Phase E - Hangup

Phase E disconnects or hangs up the modem, terminating the fax call. In FaxTransmit the input file is closed and the fax logging hook is called to indicate the end of this fax. In FaxReceive, the output file is closed and the logging hook is called.

FaxTransmit then loops back to Phase A to see whether another fax should be sent. FaxReceive waits for another incoming call unless the "one fax" option has been set.

Shared Declarations

Procedures and Functions

The routines for sending and receiving are combined in the following reference section, even though in many cases the routines come from different object types. This organization of the material seems to make it easier to find the relevant routines. Each Declaration specifies the object type in which a particular method is implemented. Each Purpose also makes clear whether the routine applies to fax send, fax receive, or both.

APFAX12 Declarations

Types

All send and receive fax records are declared using the FaxRecPtr type from the APABSFAX unit. APFAX12 declares various record types that it uses internally for FaxRecPtr variables that have been initialized for sending or receiving, but you shouldn't need to refer to any of the fields of these internal types.

OOPFAX12 Declarations

Types

```
C12AbstractFaxPtr = ^C12AbstractFax;  
C12AbstractFax =  
    object(AbstractFax)  
    ...  
end;
```

An intermediate object that defines the data and methods needed by both C12SendFax and C12ReceiveFax.

```
C12ReceiveFaxPtr = ^C12ReceiveFax;  
C12ReceiveFax =  
    object(C12AbstractFax)  
    ...  
end;
```

The object used for receiving faxes on a Class 1 or Class 2 faxmodem.

```
C12SendFaxPtr = ^C12SendFax;  
C12SendFax =  
    object(C12AbstractFax)  
    ...  
end;
```

The object used for sending faxes on a Class 1 or Class 2 faxmodem.

Syntax

destructor C12AbstractFax.Done; virtual;

Purpose

Destroy a C12AbstractFax object.

Description

This destructor undoes the work of C12AbstractFax.Init. Don't call it yourself; it is called by the destructors of objects derived from it.

See Also

C12AbstractFax.Init

Syntax

destructor C12ReceiveFax.Done; virtual;

Purpose

Destroy a C12ReceiveFax object.

Description

This routine disposes of data structures owned by the C12ReceiveFax object and then destroys the object. Note that it does *not* close or destroy the com port it is using.

See Also

C12ReceiveFax.Init

Syntax

destructor C12SendFax.Done; virtual;

Purpose

Destroy a C12SendFax object.

Description

This routine disposes of data structures owned by the C12SendFax object and then destroys the object. It destroys the AbstractFaxConverter used by the object, but it does *not* close or destroy the com port used by the object.

See Also

C12SendFax.Init

DoneC12ReceiveFax / DoneC12SendFax

Syntax

```
procedure DoneC12ReceiveFax(var FP : FaxRecPtr);
```

Purpose

Dispose of a receive fax record.

Description

This routine disposes of the fax record pointed to by FP, previously allocated by a call to InitC12ReceiveFax, as well as any other data structures used by FP. Note that it does *not* close or destroy the com port used by FP. FP is set to nil on return.

See Also

InitC12ReceiveFax

Syntax

```
procedure DoneC12SendFax(var FP : FaxRecPtr);
```

Purpose

Dispose of a send fax record.

Description

This routine disposes of the fax record pointed to by FP, previously allocated by a call to InitC12SendFax, as well as any other data structures used by FP. Note that it does *not* close or destroy the com port used by FP. FP is set to nil on return.

See Also

InitC12SendFax

Syntax

```
procedure FaxReceiveC12(FP : FaxRecPtr);  
procedure C12ReceiveFax.FaxReceive; virtual;
```

Purpose

Receive one or more fax files.

Description

FaxReceive calls FaxReceivePart repeatedly until FaxReceivePart returns faxFinished. It returns faxFinished only if 1) an error occurred, 2) a user abort routine returned True to cancel the session, or 3) a fax was successfully received when the "one fax" option was set by SetOneFax. Note that FaxReceive does not return after each fax is received unless the one fax option is set.

Prior to calling FaxReceive, your program must initialize the faxmodem and set all fax-related user hooks. A typical flow of events is the following:

1. Call GetModemClassSupport with the Reset parameter equal to True to assure the modem supports faxing and to initialize the modem to a known state.
2. If the modem supports both Class 1 and Class 2, call SetClassType to select the desired mode. (Generally, you should select Class 2 whenever the modem supports it.)
3. Activate any desired fax user hooks. The abort hook is particularly important when receiving.
4. Call FaxReceive to wait for calls and receive faxes.

Example

```
if not GetModemClassSupport(F, Class1, Class2, True) then  
  {...handle error...} ;  
if Class1 and Class2 then  
  {select Class2}  
  SetClassType(F, ctClass2)  
else if (not Class1) and (not Class2) then  
  {...handle error...};  
FaxReceiveC12(F);
```

Checks to see if the modem is a faxmodem, sets the class to Class 2, and starts waiting for faxes. When using objects, the F parameters and the C12 suffix would be omitted.

See Also

FaxReceivePart

FaxReceivePart

Syntax

```
function FaxReceivePartC12(FP : FaxRecPtr) : FaxStateType;  
function C12ReceiveFax.FaxReceivePart : FaxStateType;
```

Purpose

Perform one step of a fax receive.

Description

This function is intended for use in foreground applications or TSRs that need to multiplex fax reception with one or more other tasks. It performs one small piece of the fax receive process and then returns control to your program. In most cases the piece is so small that it returns in a fraction of a second. The only time it takes longer is when FaxReceivePart must perform a disk operation (e.g., writing a 4K block of the received image to the fax file). In that case, the time spent in FaxReceivePart depends on your hardware and operating environment.

Before you call FaxReceivePart for the first time, you must call PrepareFaxReceivePart. Note that you should also follow the general steps described under FaxReceive.

See "Background Fax Transfer" in §10.D for more information.

Example

```
PrepareFaxReceivePartC12(F);  
Finished := False;  
repeat  
  case FaxReceivePartC12(F) of  
    faxReady,  
    faxCritical : {do nothing} ;  
    faxWaiting : ProcessOtherStuff;  
    faxFinished : Finished := True;  
  end;  
until Finished;
```

Multiplexes fax reception with other tasks (ProcessOtherStuff). When using objects, the F parameters and the C12 suffixes would be omitted.

See Also

FaxReceive

PrepareFaxReceivePart

Syntax

```
procedure FaxTransmitC12(FP : FaxRecPtr);  
procedure C12SendFax.FaxTransmit; virtual;
```

Purpose

Transmit one or more fax files.

Description

FaxTransmit calls FaxTransmitPart repeatedly until FaxTransmitPart returns faxFinished. FaxTransmit doesn't return until 1) an error occurred, 2) a user abort routine returned True to cancel the session, or 3) all faxes have been transmitted.

Prior to calling FaxTransmit, your program must initialize the faxmodem, set all fax-related user hooks, and add the fax entries to be transmitted. A typical flow of events is the following:

1. Call GetModemClassSupport with the Reset parameter equal to True to assure the modem supports faxing and to initialize the modem to a known state.
2. Call SetClassType to select the transmission mode. (Generally, you should select Class 2 whenever the modem supports it.)
3. Activate any desired fax user hooks.
4. Call AddFaxEntry for each fax file to transmit.
5. Call FaxTransmit to transmit all scheduled files.

Example

```
if not GetModemClassSupport(F, Class1, Class2, True) then  
  {...handle error...};  
if Class2 then  
  {select Class2}  
  SetClassType(F, ctClass2)  
else if Class1 then  
  {select Class1}  
  SetClassType(F, ctClass1)  
else  
  {...handle error...};  
AddFaxEntry(F, '260-7151', 'FAX.APF', '');  
FaxTransmitC12(F);
```

Checks to see if the modem is a fax modem, sets the class to the highest one available, adds a fax entry to send FAX.APF to 260-7151, then transmits the fax. When using objects, the F parameters and the C12 suffix would be omitted.

See Also

FaxTransmitPart

FaxTransmitPart

Syntax

```
function FaxTransmitPartC12(FP : FaxRecPtr) : FaxStateType;  
function C12SendFax.FaxTransmitPart : FaxStateType; virtual;
```

Purpose

Perform one step of a fax transmit.

Description

This function is intended for use in foreground applications or TSRs that need to multiplex fax transmission with one or more other tasks. It performs one small piece of the fax transmit process and then returns control to your program. In most cases the piece is so small that it returns in a fraction of a second. The only time it takes longer is when FaxTransmitPart must perform a disk operation (e.g., reading another raster line from the fax image file). In that case, the time spent in FaxTransmitPart depends on your hardware and operating environment, although it's usually pretty small because only one raster line is read at a time.

Before you call FaxTransmitPart for the first time, you must call PrepareFaxTransmitPart. Note that you should also follow the general steps described under FaxTransmit.

It's very important that your program tasks do not take too long before calling FaxTransmitPart again. Read the discussion of data underflow errors in "The Fax Transfer Process" in the introduction to this section. Also see "Background Fax Transfer" in §10.D for more information.

Example

```
PrepareFaxTransmitPartC12(F);  
Finished := False;  
repeat  
  case FaxTransmitPartC12(F) of  
    faxReady,  
    faxCritical : (do nothing) ;  
    faxWaiting  : ProcessOtherStuff;  
    faxFinished : Finished := True;  
  end;  
until Finished;
```

Prepares to use FaxTransmitPart by calling PrepareFaxTransmitPart. Stays in a repeat loop calling FaxTransmitPart until faxFinished is returned. When faxWaiting is returned, ProcessOtherStuff is called to multitask the fax transmit process. When using objects, the F parameters and the C12 suffixes would be omitted.

See Also

FaxTransmit

PrepareFaxTransmitPart

Syntax

```
function GetHangupResult(FP : FaxRecPtr) : Word;
function C12AbstractFax.GetHangupResult : Word;
```

Purpose

Return the last hangup code for a Class 2 fax transfer.

Description

When a Class 2 faxmodem terminates abnormally, it returns a "hangup code" to help explain what went wrong. Unfortunately, these codes aren't particularly helpful since their descriptions are cryptic and sometimes refer to low-level portions of the faxmodem link over which you have no control. However, sometimes the hangup code may point out a programming error that you can correct.

The following table shows the codes that can be returned (in hexadecimal), with a brief description of each one. The codes are grouped according to the transfer phase in which they can occur. Some of the terms in the table are defined only in the Class 2 specification. Refer to that specification for more information.

Value	Description
Call placement and termination	
00	Normal end of connection
01	Ring detect without successful handshake
02	Call aborted from +FKS or <Can>
03	No loop current
04	Ringback detected, no answer timeout
05	Ringback detected, answer without CED
Transmit phase A	
10	Unspecified phase A error
11	No answer
Transmit phase B	
20	Unspecified phase B error
21	Remote cannot receive or send
22	COMREC error in transmit phase B
23	COMREC invalid command received
24	RSPREC error
25	DCS sent three times without response
26	DIS/DTC received three times; DCS not recognized
27	Failure to train at 2400 BPS or +FMS error
28	RSPREC invalid response received
Transmit phase C	
40	Unspecified transmit phase C error
41	Unspecified image format error
42	Image conversion error
43	DTE to DCE data underflow
44	Unrecognized transparent data command
45	Image error, line length wrong
46	Image error, page length wrong
47	Image error, wrong compression code

GetHangupResult

Transmit phase D

50	Unspecified transmit phase D error
51	RSPREC error
52	MPS sent three times without response
53	Invalid response to MPS
54	EOP sent three times without response
55	Invalid response to EOP
56	EOM sent three times without response
57	Invalid response to EOM
58	Unable to continue after PIN or PIP

Receive phase B

70	Unspecified receive phase B error
71	RSPREC error
72	COMREC error
73	T.30 T2 timeout, expected page not received
74	T.30 T1 timeout after EOM received

Receive phase C

90	Unspecified receive phase C error
91	Missing EOL after 5 seconds
92	Bad CRC or frame (ECM mode)
93	DCE to DTE buffer overflow

Receive phase D

A0	Unspecified receive phase D error
A1	RSPREC invalid response received
A2	COMREC invalid response received
A3	Unable to continue after PIN or PIP

Example

```
FaxReceiveC12(F);  
case AsyncStatus of  
  ecOk : {success} ;  
  ecFaxSessionError :  
    begin  
      WriteLn('Error during session');  
      WriteLn('Hangup result was ', GetHangupResult(F));  
    end;  
  ...  
end;
```

Displays the hangup result code if the receive session terminates with `ecFaxSessionError`. If no hangup code was returned, `GetHangupResult` returns zero.

Syntax

```
function GetLastPageStatus(FP : FaxRecPtr) : Boolean;  
function C12AbstractFax.GetLastPageStatus : Boolean;
```

Purpose

Return True if the last page was received successfully.

Description

Call this function in a fax status procedure to display whether the last page was received with or without an error. The appropriate time to call it is when GetFaxProgress returns the value fpGetPageResult.

This function's result is for information only. Regardless of what you do with GetLastPageStatus, FaxReceive automatically proceeds to the next page, requests a resend of the current page (up to the limit specified by SetMaxRetries), or terminates with an error.

Example

```
case GetFaxProgress of  
  ...  
  fpGetPageResult :  
    if GetLastPageStatus(F) then  
      WriteLn('Page accepted')  
    else  
      WriteLn('Page rejected');  
  ...  
end;
```

Part of a fax status routine that displays the status of the last received page. When using objects, the F parameter would be omitted.

GetModemClassSupport

Syntax

```
function GetModemClassSupport(FP : FaxRecPtr; var Class1, Class2 : Boolean;  
                               Reset : Boolean) : Boolean;  
function C12AbstractFax.GetModemClassSupport(var Class1, Class2 : Boolean;  
                                               Reset : Boolean) : Boolean;
```

Purpose

Determine which fax classes a modem supports.

Description

This function determines the class of the modem associated with the fax object or record. All programs using APFAX12/OOFAX12 should call it at least once to determine what classes are supported and to assure that proper initialization commands are sent to the modem.

GetModemClassSupport returns True if it could successfully determine the class support level. It returns False if the class detection failed; usually this means the modem is not a faxmodem.

Class1 returns True if the faxmodem supports the Class 1 specification. Class2 returns True if it supports Class 2. Note that both or neither of these options could be True. If the faxmodem reports that it supports both Class 1 and Class 2, you should generally use the Class 2 features.

Reset controls whether or not the modem is "reset" before testing for fax support. If Reset is True, the DTR signal is lowered for 1/2 second then raised again, to make sure the modem is in command mode. Then string specified by the Deflnit constant of APABSFAX/OOABSFAX ('ATE0Q0V1X4S0=0') is sent to the modem. This string turns command echo off, enables modem responses in word (text) format, enables the X4 response set (which causes the modem to return 'BUSY' on busy connections), and turns off auto-answer.

FaxTransmit and FaxReceive assume the modem has these settings. If you change these settings, FaxTransmit and FaxReceive probably won't work. You can safely send your own modem initialization string (e.g., in order to assure that the speaker is off) by doing so *before* you call GetModemClassSupport with Reset set to True.

Example

```
if GetModemClassSupport(F, Class1, Class2, True) then begin  
  if Class2 then  
    SetClassType(F, ctClass2)  
  else if Class1 then  
    SetClassType(F, ctClass1)  
  else  
    {...report unknown faxmodem error...}  
end else  
  {...report not a faxmodem...}
```

Checks to see what classes are supported by the modem, then enables the highest supported class. Although it doesn't happen at this time, future modems may report that they are faxmodems, and at the same time support neither Class1 or Class2. This logic checks for that possibility and reports an error. An error is also reported if the modem is not a faxmodem. When using objects, the F parameter would be omitted.

See Also

GetModemInfo

SetClassType

Syntax

```
procedure GetModemFeatures(FP : FaxRecPtr; var BPS : LongInt;  
                           var Correction : Char);  
  
procedure C12AbstractFax.GetModemFeatures(var BPS : LongInt;  
                                           var Correction : Char);
```

Purpose

Return the features supported by a Class 2 faxmodem.

Description

This procedure attempts to determine whether a Class 2 faxmodem supports error correction, and what its highest transfer rate is. It works only for Class 2 modems; a Class 1 modem cannot report this information until a fax connection has been established.

GetModemFeatures works by attempting to enable the most capable modem features and stepping down if the modem returns 'ERROR'. It starts at a 14400 BPS transfer rate, then tries 12000, 9600, 7200, 4800, and 2400. It returns one of these values in the BPS parameter. Similarly, it tries Correction = '1' to see if error correction is available. If so, it returns this character. If not, it returns '0'. (A Char result is used rather than a Boolean because the specification may support different error correction techniques in the future.)

The technique used by GetModemFeatures works on most Class 2 faxmodems. One low-cost, no-name-clone faxmodem we tested wouldn't return 'ERROR' no matter what we asked it to do, even though it supported only 9600 BPS with no error correction.

Example

```
if Class2 then begin  
  GetModemFeatures(F, BPS, ECM);  
  WriteLn('Faxmodem supports ', BPS, ' bits/second');  
  if ECM = '1' then  
    WriteLn('Faxmodem supports error correction')  
  else  
    WriteLn('Faxmodem does not support error correction');  
end;
```

Displays the transfer rate and error correction capabilities of a Class 2 modem. When using objects, the F parameter would be omitted.

See Also

GetModemClassSupport

SetModemFeatures

GetModemInfo

Syntax

```
function GetModemInfo(FP : FaxRecPtr; var Class : Char;  
    var Model, Chip, Rev : String;  
    Reset : Boolean) : Boolean;  
  
function C12AbstractFax.GetModemInfo(var Class : Char;  
    var Model, Chip, Rev : String;  
    Reset : Boolean) : Boolean;
```

Purpose

Get information about a Class 1 or Class 2 faxmodem.

Description

Set the Reset parameter to True if you want to reset the modem before querying its capabilities. See GetModemClassSupport for more information about resetting faxmodems.

This function returns True if the modem is a faxmodem, False otherwise. Don't count on any of the parameters if GetModemInfo returns False.

Class is the highest class number supported: '1' or '2'. Model, Chip, and Revision are string values returned for Class 2 modems only. They are set to empty strings for Class 1 modems.

Note that this function returns just the highest class supported. To find out if a modem supports *both* Class 1 and Class 2, use GetModemClassSupport.

Example

```
if GetModemInfo(F, Class, Model, Chip, Rev, True) then begin  
    WriteLn('Highest class: ', Class);  
    if Class = '2' then begin  
        WriteLn('Chip: ', Chip);  
        WriteLn('Model: ', Model);  
        WriteLn('Revision: ', Rev);  
    end;  
end;
```

Displays information about the faxmodem. When using objects, the F parameter would be omitted.

See Also

GetModemClassSupport

Syntax

```
procedure GetPageInfoC12(FP : FaxRecPtr; var Pages : Word; var Page : Word;  
    var BytesTransferred : LongInt;  
    var PageLength : LongInt);  
  
procedure C12AbstractFax.GetPageInfo(var Pages : Word; var Page : Word;  
    var BytesTransferred : LongInt;  
    var PageLength : LongInt); virtual;
```

Purpose

Get information about the page currently being transmitted or received.

Description

Call this routine in a fax status procedure to return information about the page currently being sent or received. The appropriate time to call it is when GetFaxProgress returns the value fpSendPage or fpGetPage. If you call it at other times, it returns either zeros or values associated with the previous page.

When a program is sending faxes, Pages contains the total number of pages in the current fax, Page is the number of the current page being transmitted, BytesTransferred is the number of bytes transmitted so far for this page, and PageLength is the total number of bytes in the current page.

When a program is receiving faxes, Pages is always zero (the total number of pages is not known in advance), Page is number of the current page being received, BytesTransferred is the number of bytes received so far for this page, and PageLength is zero (the total size of the page is not known in advance).

Example

```
...  
GetPageInfoC12(F, Pages, Page, Bytes, PageLength);  
WriteLn('Total pages: ', Pages);  
WriteLn('Current page: ', Page);  
WriteLn('Page length: ', PageLength);  
WriteLn('Bytes: ', Bytes);  
...
```

Part of fax transmit status procedure that displays information about the fax transmit in progress. When using objects, the F parameter and the C12 suffix would be omitted.

Syntax

```
function GetRemoteID(FP : FaxRecPtr) : Str20;  
function C12AbstractFax.GetRemoteID : Str20;
```

Purpose

Return the remote station ID.

Description

Call this function in a fax status procedure to display the station ID of the remote partner. The appropriate time to call it is when GetFaxProgress returns the value fpGotRemoteID. If you call it earlier, it returns an empty string.

Example

```
case GetFaxProgress of  
    ...  
    fpGotRemoteID :  
        WriteLn('Connected to ', GetRemoteID(F));  
    ...  
end;
```

Part of a status procedure that displays the remote station ID. When using objects, the F parameter would be omitted.

GetSessionParams

Syntax

```
procedure GetSessionParams(FP : FaxRecPtr; var BPS : LongInt;  
                           var Resolution : Boolean;  
                           var Correction : Boolean);  
  
procedure C12AbstractFax.GetSessionParams(var BPS : LongInt;  
                                           var Resolution : Boolean;  
                                           var Correction : Boolean);
```

Purpose

Return fax session parameters.

Description

Call this routine in a fax status procedure to return information about the negotiated transfer rate, fax resolution, and error correction mode. The appropriate time to call it is when GetFaxProgress returns the value fpSessionParams. If you call it earlier, the returned values are undefined.

Note that session parameters can change more than once during a single session. Be sure that your status routine updates its parameter display each time GetFaxProgress returns the value fpSessionParams. (In some cases you may want to save the first values returned by GetSessionParams and update the display only if they change.)

BPS is the transfer rate in bits per second. It can take on the values 14400, 12000, 9600, 7200, 4800, and 2400. Most faxmodems now support 9600 or higher. The fax connection process attempts to negotiate the highest possible rate unless you have called SetModemFeatures to limit the highest rate.

Resolution returns True for a high resolution fax transfer, or False for a standard resolution transfer. Async Professional automatically enables high resolution if it is sending an APF file that contains high resolution data, or if it is receiving a high resolution fax from a remote partner.

Correction returns True if automatic error correction is enabled for this transfer, or False if it isn't. Error correction is automatically enabled if both modems support it, unless you have called SetModemFeatures to disable the feature.

Example

```
case GetFaxProgress of  
  ...  
  fpSessionParams :  
    begin  
      GetSessionParams(F, BPS, HiRes, ECM);  
      WriteLn('Bit rate: ', BPS);  
      if HiRes then  
        WriteLn('High resolution connection')  
      else  
        WriteLn('Low resolution connection');  
      if ECM then  
        WriteLn('Error correcting mode')  
      else  
        WriteLn('Non-error correcting mode');  
    end;  
  ...  
end;
```

Part of a fax status procedure that displays the negotiated session parameters. When using objects, the F parameter would be omitted.

Syntax

```
constructor C12AbstractFax.Init(ID : Str20; ComPort : AbstractPortPtr);
```

Purpose

Initialize a C12AbstractFax object.

Description

This constructor initializes a C12AbstractFax object, which contains data fields and methods shared by C12ReceiveFax and C12SendFax. It allocates a 4K file I/O buffer used by both descendants.

Don't call this constructor directly; it is called by the descendant objects when you call their constructors.

Syntax

```
constructor C12ReceiveFax.Init(ID : Str20; ComPort : AbstractPortPtr);
```

Purpose

Initialize a C12ReceiveFax object.

Description

This constructor initializes an object for receiving faxes using a Class 1 or Class 2 faxmodem. ID is the station ID the faxmodem will report, and ComPort is an initialized serial port used for communication with the faxmodem.

ComPort must have an input buffer of at least 4096 bytes. If it does not, this constructor fails with AsyncStatus set to ecBuffersTooSmall.

Example

```
var
  UP : UartPort;
  F : C12ReceiveFax;
...
UP.InitFast(Com1, 19200);
F.Init('719-260-7151', @UP);
if AsyncStatus <> ecOk then {...handle errors...}
```

Instantiates a C12ReceiveFax object that uses Com1 at 19200 baud, and a station ID of 719-260-7151.

Syntax

```
constructor C12SendFax.Init(ID : Str20; ComPort : AbstractPortPtr);
```

Purpose

Initialize a C12SendFax object.

Description

This constructor initializes an object for sending faxes using a Class 1 or Class 2 faxmodem. ID is the station ID the faxmodem will report, and ComPort is an initialized serial port used for communication with the faxmodem.

This routine also instantiates an internal AbstractFaxConverter object used for converting cover pages and fax page header lines. The AbstractConverter requires approximately 24KB of heap space.

ComPort must have an output buffer of at least 8192 bytes. If it does not, this constructor fails with AsyncStatus set to ecBuffersTooSmall.

Example

See the example for C12ReceiveFax.Init. The only difference is that F should be declared with type C12SendFax.

InitC12ReceiveFax / InitC12SendFax

Syntax

```
procedure InitC12ReceiveFax(var FP : FaxRecPtr; ID : Str20;  
                           ComPort : PortRecPtr);
```

Purpose

Allocate and initialize a receive fax record.

Description

This procedure initializes FP to receive faxes using the specified com port and station ID.

ComPort must have an input buffer of at least 4096 bytes. If it does not, this routine fails with AsyncStatus set to ecBuffersTooSmall.

Example

```
var  
  UP : PortRecPtr;  
  F : FaxRecPtr;  
...  
  InitFast(UP, Com1, 19200);  
  InitC12ReceiveFax(F, '719-260-7151', @UP);  
  if AsyncStatus <> ecOk then  
    {...handle errors...}
```

Allocates and initializes a record for receiving faxes that uses Com1 at 19200 baud, and a station ID of 719-260-7151.

See Also

DoneC12ReceiveFax

Syntax

```
procedure InitC12SendFax(var FP : FaxRecPtr; ID : Str20; ComPort : PortRecPtr);
```

Purpose

Allocate and initialize a send fax record.

Description

This procedure initializes FP to send faxes using the specified com port and station ID.

This routine also initializes an internal FaxConverterPtr record used for converting cover pages and fax page header lines. The FaxConverterPtr requires approximately 24KB of heap space.

ComPort must have an output buffer of at least 8192 bytes. If it does not, this routine fails with AsyncStatus set to ecBuffersTooSmall.

Example

```
var  
  UP : PortRecPtr;  
  F : FaxRecPtr;  
...  
  InitFast(UP, Com1, 19200);  
  InitC12SendFax(F, '719-260-7151', @UP);  
  if AsyncStatus <> ecOk then  
    {...handle errors...}
```

Allocates and initializes a record for sending faxes that uses Com1 at 19200 baud, and a station ID of 719-260-7151.

See Also

DoneC12SendFax

Syntax

```
function InitModemForFaxReceive(FP : FaxRecPtr) : Boolean;  
function C12ReceiveFax.InitModemForFaxReceive : Boolean;
```

Purpose

Reinitialize a faxmodem for receiving faxes.

Description

This function sends appropriate 'AT' commands to the faxmodem to prepare it for receiving faxes. It first sends any string you have specified using SetModemInit. Next it sends the Definit string ('ATE0Q0V1X4S0=0') as defined by APFAX12/OOFAX12. Then it enables Class 1 or Class 2 operation, either by autodetecting the highest class supported by the modem or using the last class specified by SetClassType.

For Class 2 modems it also sends strings for setting the station ID (see SetStationID in APABSFAX/OOABSFAX), transfer rate and error correction options (see SetModemFeatures), and "fax and data" option (see SetFaxAndData). Hence, these routines must be called before calling InitModemForFaxReceive. (For Class 1 modems, these features are set while negotiating the connection, so you can call the relevant routines any time before calling FaxReceive.)

InitModemForFaxReceive returns True if all commands were successful, False otherwise. If it returns False, you can examine AsyncStatus to see what went wrong.

The critical portions of InitModemForFaxReceive are performed automatically whenever you call FaxReceive or PrepareFaxReceivePart, so you don't need to call it yourself in most situations. However, it must be called again whenever the modem has been used for any purpose besides faxing. For example, a TSR used for receiving faxes should call it whenever the modem may have been used as a data modem. The RFAX/RFAXO demonstration programs call InitModemForFaxReceive whenever you run them with the /R command line option.

InitModemForFaxReceive is also necessary when you are answering both data and fax calls on the same line. See SetFaxAndData for details.

See Also

FaxReceive

PrepareFaxReceivePart / PrepareFaxTransmitPart

Syntax

```
procedure PrepareFaxReceivePartC12(FP : FaxRecPtr);  
procedure C12ReceiveFax.PrepareFaxReceivePart;
```

Purpose

Prepare to call FaxReceivePart.

Description

Calling this routine is necessary only if your program uses FaxReceivePart instead of FaxReceive. PrepareFaxReceivePart prepares FaxReceivePart for the initial call from your program.

Example

See FaxReceivePart for an example.

Syntax

```
procedure PrepareFaxTransmitPartC12(FP : FaxRecPtr);  
procedure C12SendFax.PrepareFaxTransmitPart;
```

Purpose

Prepare to call FaxTransmitPart.

Description

Calling this routine is necessary only if your program uses FaxTransmitPart instead of FaxTransmit. PrepareFaxTransmitPart prepares FaxTransmitPart for the initial call from your program.

Example

See FaxTransmitPart for an example.

Syntax

```
procedure SetAnswerOnRing(FP : FaxRecPtr; AOR : Integer);  
procedure C12ReceiveFax.SetAnswerOnRing(AOR : Integer);
```

Purpose

Set the number of rings before a call is answered.

Description

This procedure sets the number of 'RING' responses before answering an incoming call to AOR. The default is one ring. Values less than or equal to zero are treated the same as one ring.

Example

```
SetAnswerOnRing(F, 5);
```

Tells FaxReceive or FaxReceivePart to count five rings before answering a call.

Syntax

```
function SetClassType(FP : FaxRecPtr; CT : ClassType) : ClassType;  
function C12AbstractFax.SetClassType(CT : ClassType) : ClassType;
```

Purpose

Set class of faxmodem support, returning the now current class.

Description

Valid types for CT are:

- ctDetect - the faxmodem is set to the highest compatible class
- ctClass1 - the faxmodem is set to Class 1
- ctClass2 - the faxmodem is set to Class 2
- ctCAS - used for CAS modems only
- ctUnknown - used only as a return value, if an error occurs

If CT is ctDetect, SetClassType calls GetModemClassSupport to see whether it supports Class 1, Class 2, or both. It then enables Class 2 if possible, Class 1 otherwise.

Unless CT is ctDetect, SetClassType does not validate the requested class. It simply stores it in the fax record/object.

SetClassType returns the currently selected class as its function result. It never returns ctDetect.

If you don't call this routine before sending or receiving faxes, the highest supported class is automatically enabled.

Example

```
if GetModemClassSupport(F, Class1, Class2, True) then  
  SetClassType(ctClass1)  
else  
  {...error, not a faxmodem...}
```

Forces Class 1 operation even if the faxmodem supports Class 2. When using objects, the F parameters would be omitted.

See Also

GetModemClassSupport

InitModemForFaxReceive

SetConnectState / SetDialPrefix / SetDialTime

Syntax

```
procedure SetConnectState(FP : FaxRecPtr);  
procedure C12ReceiveFax.SetConnectState;
```

Purpose

Force the receiver to pick up a connection in progress.

Description

This procedure effectively causes a jump into the middle of FaxReceive. It is intended to be used when your program answers calls that may contain either faxes or data. When your program detects an incoming fax, it should call SetConnectState and then call FaxReceive, which will start processing at the point where an incoming call has already been answered.

This approach can be used only if your faxmodem supports "adaptive answer." This term is used by modem manufacturers to indicate the ability to discriminate between fax and data calls.

Example

See SetFaxAndData for more information and an example.

Syntax

```
procedure SetDialPrefix(FP : FaxRecPtr; P : String);  
procedure C12SendFax.SetDialPrefix(P : String);
```

Purpose

Set the dial prefix string.

Description

Use this routine to specify an optional dial prefix that is inserted in the dial command between 'ATDT' and the number to dial. If your telephone system requires special numbers or codes when dialing out, you may find it more convenient to specify them once here rather than in every fax number you dial.

Do *not* include 'ATD' or a 'T' or 'P' tone/pulse modifier in the dial prefix. 'ATD' is automatically prefixed by FaxTransmit and the 'T' or 'P' or controlled by SetToneDial.

Example

```
SetDialPrefix(F, '9,');
```

Sets the dial prefix to '9,'. If tone dialing is in use, the dial command would then be 'ATDT9,<number>'. When using objects, the F parameter would be omitted.

See Also

SetDialTime

SetToneDial

Syntax

```
procedure SetDialTime(FP : FaxRecPtr; DT : Integer);  
procedure C12SendFax.SetDialTime(DT : Integer);
```

Purpose

Set the dial timeout.

Description

Use this routine to change the dial timeout—the number of ticks that FaxTransmit waits for a connection after dialing the number. The default is 1092 ticks, or 60 seconds.

Example

```
SetDialTime(F, Secs2Ticks(90));
```

Sets the dial timeout to 90 seconds. When using objects, the F parameter would be omitted.

Syntax

```
procedure SetFaxAndData(FP : FaxRecPtr; OnOff : Boolean);  
procedure C12ReceiveFax.SetFaxAndData(OnOff : Boolean);
```

Purpose

Specify whether a compatible faxmodem will answer data calls.

Description

If OnOff is True, the faxmodem is configured to answer both fax and data calls. If False, only fax calls will be answered.

Note that not all faxmodems support this feature and that there is no broad standard for enabling it. It is available only in Class 2 faxmodems. The modem must claim "adaptive answer" or a similar term in its list of features. It must accept the 'AT+FAA=1' sequence that Async Professional sends to enable the feature. The "SupraFAXmodem" is one modem that meets all of these requirements.

To use this feature, you should first call SetFaxAndData with OnOff set to True. Then call InitModemForFaxReceive to initialize the modem for receiving either fax or data calls. Do *not* call FaxReceive. Instead, wait for incoming calls in your own code. When a call arrives and you issue the 'ATA' command, the faxmodem will detect whether the call is data or fax.

If the call is data, the modem will respond with 'CONNECT'. If you detect this string, your program should proceed as it normally does for a data call.

If the call is fax, the modem will respond with 'CED', 'FAX', or '+FCON'. If you detect any of these strings, call SetConnectState to tell FaxReceive that the call has already been answered, then call FaxReceive.

This area of faxmodem behavior has not been standardized. You may need to experiment with your brand of faxmodem to find a method that works for you.

Example

```
WaitForCall(Fax);  
if Fax then begin  
  SetConnectState(F);  
  FaxReceiveC12(F);  
end else begin  
  {...handle data call...}  
end;
```

WaitForCall is a procedure in your code that waits for incoming calls, answers them, and sets Fax to True if the call is a fax, False otherwise. This example then shows how to process the call if it is a fax. When using objects, the F parameters and the C12 suffix would be omitted.

See Also

InitModemForFaxReceive

SetConnectState

SetFaxPort

Syntax

```
procedure SetFaxPort(FP : FaxRecPtr; ComPort : PortRecPtr);  
procedure C12AbstractFax.SetFaxPort(ComPort : AbstractPortPtr);
```

Purpose

Specify a new com port record/object.

Description

Call this routine to change the com port used by the fax record/object. You need to use it only if the com port has changed since you initialized the fax record/object.

Example

```
InitPortFast(ComPort, Com1, 19200);  
InitC12ReceiveFax(F, 'STATION ID', ComPort);  
FaxReceiveC12(F);  
...  
DonePort(ComPort);  
InitPortFast(ComPort, Com2, 19200);  
SetFaxPort(F, ComPort);
```

The fax record initially uses Com1 for fax reception. Later, it is changed to use Com2 instead.

```
New(ComPort, InitFast(Com1, 19200));  
New(F, Init('STATION ID', ComPort));  
F^.FaxReceive;  
...  
Dispose(ComPort, Done);  
New(ComPort, InitFast(Com2, 19200));  
F^.SetFaxPort(ComPort);
```

The OOP equivalent of the previous example.

Syntax

```
procedure SetHeaderText(FP : FaxRecPtr; S : String);  
procedure C12SendFax.SetHeaderText(S : String);
```

Purpose

Specify a fax header line.

Description

Call this procedure to specify an optional header line that is sent at the top of each fax page. If you don't call it, no header line is transmitted.

A header line consists of normal text and replacement tags. A replacement tag is one of several strings prefixed with '\$', as defined below. When the header line is transmitted, the tags are replaced with appropriate text. The following replacement tags are provided:

- \$D - today's date in MM/DD/YY format, always 8 characters
- \$I - station ID
- \$N - total number of pages, variable size
- \$P - current page number, variable size
- \$R - recipient's name as set by SetRecipientName, variable size
- \$F - sender's name as set by SetSenderName, variable size
- \$S - title as set by SetTitle, variable size
- \$T - current time in HH:MMpm format, always 7 characters

Note that some of the tags vary in length. For example, \$P would be replaced by '1' for the first page and '10' for the tenth page.

SetHeaderText does not check to make sure your line fits on a page. If your line does not fit, it is truncated when the header line is transmitted. Using the default fonts, you can fit approximately 85 characters on a standard width page.

See "Cover Pages" in §10.D for more information.

Example

```
const  
  Header = '$D $T From: $S $I Page $P of $N';  
...  
SetTitle(F, 'Demo');  
SetHeaderText(F, Header);
```

Specifies a header consisting of both header tags and normal text. When the replacement tags are filled in, this header would look something like the following:

```
'01/01/94 15:00am From: Demo 719 260 7151 <28 spaces> Page 1 of 2'
```

When using objects, the F parameters would be omitted.

See Also

SetTitle

SetMaxRetries

Syntax

```
procedure SetMaxRetries(FP : FaxRecPtr; MR : Integer);  
procedure C12SendFax.SetMaxRetries(MR : Integer);
```

Purpose

Set the maximum number of retries for transmitting a fax.

Description

Use this routine to specify the maximum number of times FaxTransmit attempts to send a fax page. The default is two, which means that FaxTransmit will retry twice, for at most three attempts at sending each page. If you don't want FaxTransmit to retry at all, pass zero for MR.

If all attempts to send a page fail, FaxTransmit returns with AsyncStatus set to ecFaxSessionError.

Example

```
SetMaxRetries(F, 1);
```

Enables a single retry when a page error occurs.

Syntax

```
procedure SetModemFeatures(FP : FaxRecPtr; BPS : LongInt; Correction : Char);  
procedure C12AbstractFax.SetModemFeatures(BPS : LongInt; Correction : Char);
```

Purpose

Set desired modem features for a fax session.

Description

Use this routine to specify the highest allowed data transfer rate and error correction mode for the next fax session. Acceptable values for BPS are 14400, 12000, 9600, 7200, 4800, and 2400. Acceptable values for Correction are '1' (enable error correction) and '0' (disable it). The default values are 9600 and '0'.

Because of how fax connections are negotiated, asking for a particular feature doesn't mean that you'll get it. The final transfer rate and error correction mode will be determined by the actual capabilities of the local and remote modem as well as the telephone line conditions. In general, however, you shouldn't ask for a feature that you know your modem doesn't support, because you may make an unreliable connection.

Calling SetModemFeatures does not cause any faxmodem commands to be issued. It just sets fields of the fax record/object that are used during the negotiation phase of a fax call. Hence, unlike GetModemFeatures, it can be used with both Class 1 and Class 2 faxmodems.

Example

```
SetModemFeatures(F, 14400, '1');
```

Requests a 14400 BPS transfer rate with error correction. When using objects, the F parameter would be omitted.

See Also

GetModemFeatures

Syntax

```
procedure SetModemInit(FP : FaxRecPtr; MIS : String);  
procedure C12AbstractFax.SetModemInit(MIS : String);
```

Purpose

Specify a custom modem initialization string.

Description

If you specify a custom modem initialization string, Async Professional always sends this string to the modem just before it sends its own DefInit string ('ATE0Q0V1X4S0=0'). This occurs whenever you call FaxTransmit, FaxReceive, PrepareFaxTransmitPart, PrepareFaxReceivePart, InitModemForFaxReceive, or GetModemClassSupport (with the Reset parameter set to True).

A custom initialization string is sometimes needed to enable one-way software flow control between the faxmodem and the PC. Unfortunately the string used for this purpose varies from modem to modem. See MODEM.DOC in your VAPRO directory for information that we have gleaned about various faxmodems. If you have additions to this file, we would like to hear them.

Note that the DefInit string may override certain actions of the SetModemInit string. This is necessary for proper operation of the Async Professional fax routines.

MIS should not contain an 'AT' prefix or a trailing carriage return.

Example

```
SetModemInit(F, 'M0');
```

Turns off the faxmodem speaker. When using objects, the F parameter would be omitted.

See Also

GetModemClassSupport

SetOneFax / SetToneDial

Syntax

```
procedure SetOneFax(FP : FaxRecPtr; OnOff : Boolean);  
procedure C12ReceiveFax.SetOneFax(OnOff : Boolean);
```

Purpose

Enable or disable "one fax" receive behavior.

Description

The setting you specify by calling SetOneFax affects the behavior of FaxReceive and FaxReceivePart. When you pass True to SetOneFax, FaxReceive returns after receiving one fax. Similarly, FaxReceivePart returns a result of faxFinished after receiving one fax. By default, or if you pass False to SetOneFax, FaxReceive doesn't return until a fatal error is detected or a user abort hook forces an exit. And FaxReceivePart doesn't return a result of faxFinished until one of these events occurs.

The most likely use of "one fax" behavior is in conjunction with SetFaxAndData and SetConnectState. These are used when your program handles incoming calls itself and passes control to the fax routines only after a fax call has been detected.

Example

```
SetOneFax(F, True);  
FaxReceiveC12(F);
```

FaxReceive accepts just one fax before returning control to your program. When using objects, the F parameters and the C12 suffix would be omitted.

See Also

SetConnectState

SetFaxAndData

Syntax

```
procedure SetToneDial(FP : FaxRecPtr; Tone : Boolean);  
procedure C12SendFax.SetToneDial(Tone : Boolean);
```

Purpose

Select tone or pulse dialing for fax transmissions.

Description

Set Tone to True for tone dialing, or False for pulse. This does not immediately issue a modem command, but determines whether 'T' or 'P' is added to the dial command later.

The default is tone dialing.

Example

```
SetToneDial(F, False);
```

Instructs FaxTransmit to use pulse dialing. When using objects, the F parameter would be omitted.

F. CAS Fax Send/Receive: APFAXCAS/OOFAXCAS

APFAXCAS/OOFAXCAS build onto APABSFAX/OOABSFAX to provide access to the fax send and receive services of a CAS faxmodem driver. CAS refers to the "Communicating Applications Specification" developed by DCA and Intel, which is used by the Intel SatisFAXtion line of modems. CAS actually provides data send and receive services in addition to fax, but those services are not accessible using the APFAXCAS/OOFAXCAS units.

Modems that support CAS include a driver named CASMODEMEXE that you normally load from your AUTOEXEC.BAT file. CASMODEM is responsible for converting documents to its internal fax format, and for sending and receiving faxes. Application software communicates with CASMODEM by making interrupt \$2F calls. Requests to the CASMODEM driver occur at a fairly high level, for example, "send BIG.DOC to 260-7151" or "start receiving faxes." CASMODEM queues the requests and performs all operations in the background, allowing your application to move on to other tasks.

APFAXCAS/OOFAXCAS provide two important advantages in a CAS environment. First, they provide convenient Pascal shells around the CAS fax functions, freeing you from having to figure out the int \$2F calls. Second, they fit the most important CAS calls into the framework of APABSFAX/OOABSFAX, which allows your application to support Class 1, Class 2, and CAS faxmodems with one set of source code.

Because the use of CAS is so different from direct programming of Class 1/2 faxmodems, the ability of APFAXCAS/OOFAXCAS to emulate the Async Professional fax architecture and still access CAS features is limited. Basically, APABSFAX/OOABSFAX allow you to initialize a fax record/object, call FaxTransmit or FaxReceive, activate abort, status, or logging hooks, and dispose of the record/object. The differences between using CAS and Class 1/2 are discussed later in this introduction.

APFAXCAS/OOFAXCAS also provide direct access to CAS services, which are used quite differently from Class 1/2 services. These routines are described in the reference section. For more background about them, you'll want to read the actual CAS specification. A copy of Intel's official file, CASxx.COM (where xx is the current version number), is found in your Async Professional BONUS directory. It is a self-extracting archive.

Before you can use APFAXCAS/OOFAXCAS, you must configure and install the CASMODEM driver that is provided with your CAS faxmodem. Among other things, this setup process tells CASMODEM which serial port to use. Application programs cannot alter this port setup. Hence, none of the APFAXCAS/OOFAXCAS initialization routines accept a port pointer.

How CAS Works

CAS manages fax and data transfer by using control files that are placed in one of three different queues. As events occur and tasks are completed, CAS moves the files from queue to queue. The three queues are:

- Task Queue - pending send and receive events
- Receive Queue - completed receive events (fax or data files)
- Log Queue - all completed events (successful or with errors)

When you submit a fax to send, APFAXCAS/OOFAXCAS formats an appropriate control file and submits it to CAS. CAS places the control file in the Task Queue. The control file contains date/time fields that tell CAS when to process the event. At the specified time, CAS removes it from the Task Queue and starts processing it, at which point it is called the current event. When the

current event completes (successfully or with errors) it is moved to the Log Queue. It remains there until you remove it.

When it is receiving faxes, the CAS driver operates in a totally automatic fashion; you merely enable or disable fax reception. When reception is enabled, CAS answers incoming calls and receives faxes. For each received fax, it creates a control file which it places in both the Receive Queue and Log Queue. At this point the received fax file isn't available to your application. You must extract the received file from the Receive Queue, giving it a file name and destination. This creates a DCX file, which is a multi-page PCX format that CAS uses to store fax images. The final step is to delete the received file from the Receive Queue.

Using FaxTransmit and FaxReceive

APFAXCAS/OOFAXCAS take advantage of the CAS operations just described when you call FaxTransmit or FaxReceive. This section describes how they fit into the Async Professional fax architecture.

When you specify a fax to transmit, the parameters you pass to AddFaxEntry (or return with a next fax hook) must meet different requirements for CAS transfers than they do for Class 1 and 2. For CAS, the file to be faxed must be a text file, a PCX file, or a DCX file. It should *not* be in APF format. CAS automatically converts text files to fax format using its own fonts. The cover page file for CAS must always be a text file. It cannot be in APF format. Async Professional does perform replacement tag substitution on this file before submitting it to CAS. See "Cover Pages" in §10.D for more information about replacement tags.

FaxTransmit first calls the next fax function to get the name and number of the next file to fax. It then calls your fax logging hook with the lfaxTransmitStart parameter. Next, it submits the fax job to the CAS driver. Immediately thereafter, it calls the logging hook again, using the lfaxTransmitOk parameter if the CAS driver accepted the job, or lfaxTransmitFail if CAS rejected it. At this point, CAS hasn't actually transmitted the fax, so it's only known that CAS successfully queued the job.

FaxTransmit then waits in a loop, continually calling a CAS function that checks the status of the current transfer. Within this loop, it calls the user abort and fax status hooks that you have installed. You can call the same status functions within your status routine that you would for a Class 1/2 transfer, APCASFAX/OOCASFAX fill in all the relevant information that is available from CAS. Note, however, that your status routine is called only about once per second (or at the rate specified by the DefStatusTimeout constant in APABSFAX/OOABSFAX). It is not called for each fax "state change" as occurs for a Class 1/2 transfer. Also note that, if your abort function returns True, FaxTransmit exits immediately but it does not tell CAS to abort the current fax or any other scheduled faxes. You can use CAS-specific functions outside of FaxTransmit to do so if you like.

FaxReceive performs two major activities when you call it in a CAS environment. First, it checks the CAS Receive Queue for faxes that have *already* been received. If there are any, it calls the Async Professional fax name function to get a name for the received file and then calls the fax logging function with the lfaxReceiveStart parameter and this file name. It extracts the file from the queue, stores it to the specified file, and deletes the queue entry. It then calls the fax logging function, using the lfaxReceiveOk parameter if this process succeeded, or lfaxReceiveFail if it failed. This process is repeated until the Receive Queue is empty.

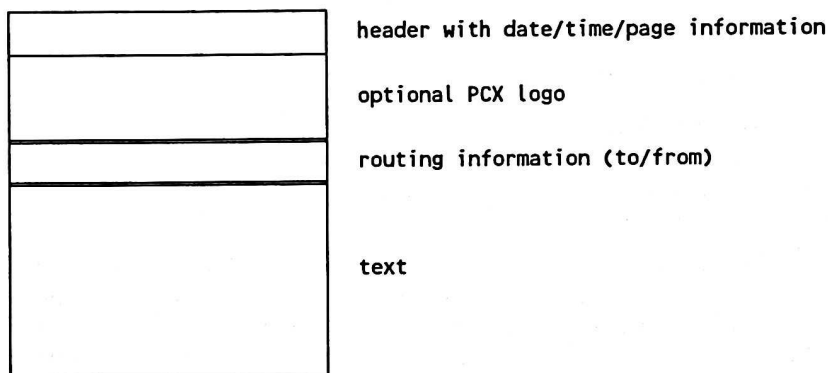
Second, FaxReceive enters another loop where it watches for incoming faxes. To do so, it attempts to get the status of the current CAS event. If it finds one, it calls the user abort and fax status hooks that you have installed, again with a one second interval for the status routine. As with FaxTransmit, it uses a CAS status function to initialize the status fields likely to be needed by your

routine. When the receive event is complete, the file is extracted from the Receive Queue as described above.

You need to watch out for side effects of the CAS background operation when you are using FaxTransmit and FaxReceive. To safely use FaxTransmit, you should send *all* of your application's faxes using FaxTransmit. Otherwise, if a fax is already being sent when you call FaxTransmit, the routine will confuse this current event with the one that you are attempting to schedule. Similarly, you should disable CAS fax receives before you call FaxTransmit so that an incoming fax won't get processed just before FaxTransmit attempts to send your fax.

CAS Cover Pages

The CAS routines provide both more and less in the way of cover pages than the APABSFAX/OOABSFAX routines. More, because cover pages can include a PCX logo file and less, because your program has far less control over the formatting of the cover page. Here's the CAS layout for a cover page:



The header uses the same format as the one that appears at the top of each page. CAS doesn't document any options for modifying the header.

The PCX logo file is set by SetLogoFile. It is included in faxes transmitted by SubmitSingleFile and FaxTransmit only when the afCASSubmitUseControl option is enabled. The maximum size of the PCX logo image is 1712 pixels wide by 800 pixels long (about 8.5 by 4 inches).

The routing information block always consists of a top and bottom border using the double line drawing character with the following four information fields:

To: <SetRecipientName>	Date: <today's date>
From: <SetSenderName>	Page: 1 of <total>

The To and From fields are filled in from the values passed to the SetRecipientName and SetSenderName procedures, respectively. The Date field is filled in by CAS with the current date. The total pages field is also filled in by CAS. The CAS specification does not document a method for modifying the routing information block.

The final area of the cover page is the text area. CAS places your specified text in this block. When using FaxTransmit, the text information comes from the cover text file set by AddFaxEntry or

returned by your NextFax function. When using SubmitSingleFile, the text information comes from the specified cover text file.

The cover text file can contain spaces and line breaks to provide the desired formatting. It can also contain replacement tags as described in "Cover Pages" in §10.D. However, most of those replacement tags are designed to provide routing information, which CAS has already provided in its routing information block. So, you are less likely to need these tags in your text block. Here are the tags available to CAS cover pages:

- \$D - today's date in MM/DD/YY format, always 8 characters
- \$I - station ID, variable
- \$R - recipients name as set by SetRecipientName, variable
- \$F - sender's name as set by SetSenderName, variable
- \$S - title as set by SetTitle, variable
- \$T - current time in HH:MMpm format, always 7 characters

FaxTransmit and SubmitSingleFile use the presence or absence of a cover text file as the indication of whether or not you want a cover file. If a cover text file is supplied, a cover page is sent; if a cover text file is not supplied, no cover page is sent. If you want a cover file without any text (just a logo and routing information), you must supply an empty cover file.

The CAS User Abort Hook

The user abort hook of a Class 1 or Class 2 fax transfer is actually the user abort hook of the serial port record/object being used to transmit the data. Since no such serial port record is used during CAS transfers, this hook cannot be used.

Instead, APFAXCAS/OOFAXCAS provide a routine named SetCASAabortFunc that can be used to install an abort hook specifically for CAS transfers. The function you pass to SetCASAabortFunc is declared exactly the same as the normal serial port abort hook, i.e., global, far, and a Boolean function that takes no parameters. It can check the keyboard for user requests just the same as a normal hook. It should *not* check any serial port status, such as Data Carrier Detect, since the CAS driver is in charge of the serial port.

See "User Abort Function" in §4.B for more information and examples.

The CAS Fax Status Hook

A fax status hook is installed for CAS in just the same way as it is for a Class 1 or Class 2 transfer. See "The Fax Status Hook" in §10.D for additional background.

Unfortunately, CAS provides much less step-by-step visibility into the progress of a fax transfer than Class 1 and 2 provide. Whereas APFAX12/OOFAX12 provide over a dozen different progress states, CAS provides only three: fpWaiting, fpSendPage, and fpGetPage.

The information returned by the GetPageInfo status function must be treated differently for CAS than for Class 1 and 2. The only parameter that is exactly the same is Pages, the total number of pages to be transmitted. The Page parameter returns the current page number. The BytesTransferred parameter returns the total bytes transferred instead of the bytes transferred for the current page. The PageLength parameter always returns zero.

Obviously, the status hook of a program intended to work with Class 1, Class 2, and CAS faxmodems needs to take these differences into account. The SIMPSND/SIMPSNDO and SIMPRCV/SIMPRCVO demonstration programs provide examples of how to do this.

CAS Error Codes

CAS defines several dozen error codes that can be returned by the CAS driver. New Async Professional error codes have not been invented to correspond to each of these CAS-specific codes. Instead, the CAS codes themselves are returned in AsyncStatus. Since there is some overlap between these codes and the native Async Professional codes, be careful to use the CAS interpretations only after you make a call to a routine in APFAXCAS or OOFAXCAS.

The following table shows (in hexadecimal) the code stored in AsyncStatus for each CAS-specific error. Note that these are transformed slightly from the codes returned by the CAS driver, which always sets the \$8000 bit. See the CAS specification for more details.

CAS Fax Error Codes

\$0000	No error
\$0002	Bad scanline count
\$0003	Page sent with errors, couldn't resend
\$0004	Receive data lost
\$0005	Invalid or missing logo file
\$0006	File name doesn't match nonstandard format (NSF) header
\$0007	File size doesn't match nonstandard format (NSF) header
\$0101	Invalid function number
\$0105	Access denied
\$0106	Invalid handle
\$0200	Multiplex handler failed
\$0201	Unknown command (bad function number)
\$0202	Event not found (bad event handle)
\$0203	Attempted to Find Next before Find First
\$0204	No more events
\$0207	Invalid Queue type (bad Queue number)
\$0208	Bad Control File
\$0209	Communication board is busy
\$020A	Invalid command parameter
\$020B	Can't uninstall the Resident Manager
\$020C	File already exists
\$0280	Unknown task type (not a Send or Poll event)
\$0281	Bad phone number
\$0282	Bad PCX file header
\$0283	Unexpected EOF
\$0284	Unexpected disconnect
\$0285	Exceeded maximum dialing retries
\$0286	No files specified for Send event
\$0287	Communication board timeout
\$0288	Received more than 1023 (maximum) pages of data
\$0289	Manual connect posted too long ago
\$028A	Hardware command set error
\$028B	Bad nonstandard format (NSF) header file
\$0302	File not found
\$0303	Path not found
\$0401	Remote unit not Group 3 compatible
\$0402	Remote unit didn't send its capabilities
\$0403	Remote unit requested disconnect
\$0404	Remote unit isn't capable of file transfers
\$0405	Exceeded retrain or fax resend limit
\$0406	Line noise or the local and remote unit don't agree on a bit rate
\$0407	Remote unit disconnected after receiving data
\$0408	No response from remote unit after sending data
\$0409	Capabilities of remote unit aren't compatible
\$040A	No dial tone - check phone line and cord (V1.2)

\$040B Invalid response from remote unit after sending data
 \$040D Phone line dead or remote unit disconnected
 \$040E Timeout while waiting for secondary dial tone (V1.2)
 \$0411 Invalid command from remote after receiving data
 \$0415 Tried to receive from incompatible hardware
 \$041F Unexpected end of file while receiving
 \$045C Received data overflowed input buffer
 \$045D Remote hardware unexpectedly stopped sending data
 \$045E Remote hardware didn't send any data
 \$045F Remote hardware took too long to send fax scan line
 \$0463 Can't get through to remote unit
 \$0464 User canceled event

Shared Declarations

Constants

MaxCoverData = 2048;

Maximum size, in bytes, of the cover text file after any substitutions are performed. Any data beyond this limit is discarded.

Types

ControlFileRecord = record

...

end;

CAS Control File Record. This record is returned as part of a StatusRecord when you call GetEventStatus, a native CAS function. You'll need to read the CAS specification to understand the two dozen fields in this record. It contains information such as the type of event to process, the time a transmission should start, and the phone number of a fax recipient.

ExternalDataBlock = record

Major	: Byte;	{CAS major version number}
Minor	: Byte;	{CAS minor version number}
Path	: Array[0..67] of Char;	{Directory of CAS driver/support files}
PB	: Array[0..12] of Char;	{Default phonebook file name}
Logo	: Array[0..12] of Byte;	{Default logo file name}
Sender	: Array[0..31] of Char;	{Default sender name}
CSID	: Array[0..20] of Char;	{Default station ID}
Reserved	: Array[0..106] of Byte;	{For future expansion}

end;

CAS External Data Block record, returned by GetExternalDataBlock, another native CAS function. All of the character arrays contain null-terminated strings. See the CAS specification for further details.

FileTransferRecord = record

FileType	: Byte;	{Type of file: text, PCX, etc.}
TextSize	: Byte;	{Page size for text files}
Status	: Byte;	{File status}
BytesSent	: LongInt;	{Bytes transferred so far}
SizeTotal	: LongInt;	{Total file size in bytes}
PagesSent	: Word;	{Pages transferred so far}
PagesTotal	: Word;	{Total pages in fax}
Path	: Array[0..79] of Char;	{Full pathname of file, ASCIIIZ}
Increments	: Byte;	{Fractional page length}
PageLen	: Byte;	{Page length code}
Reserved	: Array[0..30] of Byte;	{For future expansion}

end;

CAS File Transfer Record, not including the cover data. This record is returned as part of a StatusRecord when you call GetEventStatus, a native CAS function. A CAS control file is

composed of a ControlFileRecord, followed by one or more FileTransferRecords. See the CAS specification for more information.

```
QueueType = (qTask, qReceive, qLog);
```

Types of CAS queues.

```
StatusArray = array[0..127] of Char;
```

Array for vendor specific hardware status. See GetHardwareStatus.

```
StatusRecord = record
```

```
  CFRec          : ControlFileRecord;
```

```
  FTRec          : FileTransferRecord;
```

```
end;
```

CAS status record returned by GetEventStatus.

APFAXCAS Declarations

Types

All send and receive fax records are declared using the FaxRecPtr type from the APABSFAX unit. APFAXCAS typecasts this internally to a type that contains CAS-specific information, but you shouldn't need to refer to any of the fields of this internal type.

OOFAXCAS Declarations

Types

```
CASFaxPtr = ^CASFax;
```

```
CASFax =
```

```
  object(AbstractFax)
```

```
  ...
```

```
end;
```

The object used for sending and receiving faxes using CAS services.

AbortCurrentEvent / CASInstalled / CloseFile

Syntax

```
procedure AbortCurrentEvent(FP : FaxRecPtr; var Handle : Word);  
procedure CASFax.AbortCurrentEvent(var Handle : Word);
```

Purpose

Abort the current CAS event.

Description

This routine is a shell around CAS function \$01. It aborts the currently executing CAS event. The CAS specification notes that it may take up to 30 seconds before the current event actually aborts; however, this procedure always returns immediately.

If the abort request was successful, AsyncStatus is set to ecOk and Handle is the event handle of the aborted event; otherwise AsyncStatus contains a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
SubmitSingleFile(F, ...);  
...  
AbortCurrentEvent(F, H);
```

Submits a single file to fax, then later aborts the event, assuming that it is still the current event. When using objects, the F parameters would be omitted.

Syntax

```
function CASInstalled : Boolean;
```

Purpose

Return True if the CAS driver is installed.

Description

Your program should call this function to ensure that the CAS driver TSR is installed before it uses any other CAS functions.

Syntax

```
procedure CloseFile(FP : FaxRecPtr; FHandle : Word);  
procedure CASFax.CloseFile(FHandle : Word);
```

Purpose

Close an open file handle.

Description

This is a shell around the standard DOS call to close a file. Handle is the file handle that was returned by the CAS OpenFile routine. AsyncStatus is set to ecOk if successful, a DOS error code otherwise.

Example

```
OpenFile(F, qReceive, 1, Handle);  
...  
CloseFile(F, Handle);
```

Opens the first received file in the receive queue; later closes it.

See Also

OpenFile

Syntax

```
procedure DeleteAllFiles(FP : FaxRecPtr; Q : QueueType);  
procedure CASFax.DeleteAllFiles(Q : QueueType);
```

Purpose

Delete all files in the specified queue.

Description

This routine is a shell around CAS function \$09. It deletes all control files in queue Q. If the specified queue is the receive queue (qReceive), all received fax files are deleted as well.

AsyncStatus is set to ecOk if this operation is successful, a CAS error code otherwise. See the list of CAS error codes earlier in this section.

Example

```
DeleteAllFiles(F, qReceive);
```

Deletes all control files and all received files in the receive queue.

See Also

DeleteFile

Syntax

```
procedure DeleteFile(FP : FaxRecPtr; Handle : Word; FileNumber : Word;  
                    Q : QueueType);  
procedure CASFax.DeleteFile(Handle : Word; FileNumber : Word;  
                           Q : QueueType);
```

Purpose

Delete a file from the specified queue.

Description

This routine is a shell around CAS function \$08. It deletes one or more files from queue Q.

Handle is the event handle associated with the file or files to delete. FileNumber applies to receive queues only. If FileNumber is zero, all files associated with the event handle are deleted, otherwise FileNumber is the number of the file to delete. Q is the queue type to process.

AsyncStatus is set to ecOk if successful, a CAS error number otherwise. See the list of CAS error codes earlier in this section.

Example

```
DeleteFile(F, Handle, 1, qReceive);
```

Delete the first received file associated with the event Handle. When using objects, the F parameter would be omitted.

See Also

DeleteAllFiles

Done / DoneCASFax

Syntax

destructor CASFax.Done; virtual;

Purpose

Destroy a CASFax object.

Description

This routine disposes of data structures owned by the CASFax object and then destroys the object.

Example

See CASFax.Init for an example.

Syntax

procedure DoneCASFax(var FP : FaxRecPtr);

Purpose

Dispose of a CAS fax record.

Description

This routine disposes of data structures owned by the CAS fax record and then disposes of FP itself. FP is set to nil on return.

Example

See InitCASFax for an example.

Syntax

```
procedure FaxReceiveCAS(FP : FaxRecPtr);  
procedure CASFax.FaxReceive; virtual;
```

Purpose

Receive one or more fax files.

Description

Since the CAS driver handles the details of receiving faxes, FaxReceive's primary responsibilities are to check to see whether CAS has received or is currently receiving a fax, and to call the fax user hooks at the appropriate times.

This process is described in "Using FaxTransmit and FaxReceive" in the introduction to this section.

Note that the afCASWaitTillDone option (APABSFAX/OOFAXCAS) also affects the operation of FaxReceive. If this option is turned off (it is on by default), FaxReceive extracts fax files that are already in the receive queue but returns to the application without waiting to receive additional faxes.

Also note that fax files received by CAS are stored in DCX (multi-page PCX) format, not APF format.

Example

```
if CasInstalled then begin  
  InitCASFax(F, 'STATION ID');  
  {...set hooks...}  
  Rings := 1;  
  GetSetAutoReceive(F, Rings, 1);  
  FaxReceiveCAS(F);  
end;
```

Assures that the CAS manager is installed, initializes a CAS fax record, turns on automatic receive, and waits for incoming faxes.

```
if CasInstalled then begin  
  New(F, Init('STATION ID'));  
  {...set hooks...}  
  Rings := 1;  
  F^.GetSetAutoReceive(Rings, 1);  
  F^.FaxReceive;  
end;
```

The OOP equivalent of the previous example.

See Also

FaxReceivePart

FaxReceivePart

Syntax

```
function FaxReceivePartCAS(FP : FaxRecPtr) : FaxStateType;  
function CASFax.FaxReceivePart : FaxStateType;
```

Purpose

Perform one step of a CAS fax receive.

Description

This function is intended for use in foreground applications or TSRs that need to multiplex fax reception with one or more other tasks. It performs one small piece of the fax receive process and then returns control to your program. In most cases the piece is so small that it returns in a fraction of a second. The only time it takes longer is when FaxReceivePart must call a lengthy CAS function (e.g., MoveReceivedFile or DeleteAllFiles). In that case, the time spent in FaxReceivePart depends on your hardware and operating environment.

Before you call FaxReceivePart for the first time, you must call PrepareFaxReceivePart. Note that you should also follow the general steps described under FaxReceive.

The function result returns only two of the possible FaxStateType values: faxWaiting and faxFinished. faxFinished is returned only when FaxReceivePart is completely finished, either due to an error or a user abort request. faxWaiting is returned in all other cases.

See "Background Fax Transfer" in §10.D for more information. Note, however, that many of the timing sensitivities of Class 1/2 fax transfers do not apply to CAS, since the CAS manager is always doing the low level work in the background.

Example

```
PrepareFaxReceivePartCAS(F);  
Finished := False;  
repeat  
  case FaxReceivePartCAS(F) of  
    faxWaiting : ProcessOtherStuff;  
    faxFinished : Finished := True;  
  end;  
until Finished;
```

Multiplexes fax reception with other tasks (ProcessOtherStuff). When using objects, the F parameters and the CAS suffixes would be omitted.

See Also

FaxReceive

PrepareFaxReceivePart

Syntax

```
procedure FaxTransmitCAS(FP : FaxRecPtr);  
procedure CASFax.FaxTransmit; virtual;
```

Purpose

Transmit one or more fax files.

Description

Since the CAS driver does the actual transmission, FaxTransmit's primary responsibilities are to submit the files to CAS, and to call the fax user hooks at the appropriate times.

This process is described in "Using FaxTransmit and FaxReceive" in the introduction to this section.

Note that the afCASWaitTillDone option (APABSFAX/OOFAXCAS) also affects the operation of FaxTransmit. If this option is turned off (it is on by default), FaxTransmit submits the files and returns immediately thereafter.

Example

```
if CasInstalled then begin  
  InitCASFax(F, 'STATION ID');  
  {...set hooks...}  
  Rings := 0;  
  GetSetAutoReceive(F, Rings, 1);  
  AddFaxEntry(F, '260-7151', 'FAX1.TXT', 'COVER.TXT');  
  AddFaxEntry(F, '555-1212', 'FAX2.TXT', '');  
  FaxTransmitCAS(F);  
end;
```

Assures that the CAS manager is installed, initializes a CAS fax record, turns off automatic receive, and adds two files to be transmitted. Note that, unlike Class 1 and Class 2 transfers, the files to be faxed must be text files or PCX files, not APF files. FaxTransmit submits these two files to the CAS driver, monitors the status until both files have been sent, then returns to the program.

```
if CasInstalled then begin  
  New(F, Init('STATION ID'));  
  {...set hooks...}  
  Rings := 0;  
  F^.GetSetAutoReceive(Rings, 1);  
  F^.AddFaxEntry('260-7151', 'FAX1.TXT', 'COVER.TXT');  
  F^.AddFaxEntry('555-1212', 'FAX2.TXT', '');  
  F^.FaxTransmit;  
end;
```

The OOP equivalent of the previous example.

See Also

FaxTransmitPart

FaxTransmitPart

Syntax

```
function FaxTransmitPartCAS(FP : FaxRecPtr) : FaxStateType;  
function CASFax.FaxTransmitPart : FaxStateType;
```

Purpose

Perform one step of a CAS fax transmit.

Description

This function is intended for use in foreground applications or TSRs that need to multiplex fax transmission with one or more other tasks. It performs one small piece of the fax transmit process and then returns control to your program. In most cases the piece is so small that it returns in a fraction of a second. The only time it takes longer is when FaxTransmitPart must call a lengthy CAS function (SubmitSingleFile is the only one). In that case, the time spent in FaxTransmitPart depends on your hardware and operating environment.

Before you call FaxTransmitPart for the first time, you must call PrepareFaxTransmitPart. Note that you should also follow the general steps described under FaxTransmit.

The function result returns only two of the possible FaxStateType values: faxWaiting and faxFinished. faxFinished is returned only when FaxTransmitPart is completely finished, either due to an error or a user abort request. faxWaiting is returned in all other cases.

See "Background Fax Transfer" in §10.D for more information. Note, however, that many of the timing sensitivities of Class 1/2 fax transfers do not apply to CAS, since the CAS manager is always doing the low level work in the background.

Example

```
PrepareFaxTransmitPartCAS(F);  
Finished := False;  
repeat  
  case FaxTransmitPartCAS(F) of  
    faxWaiting : ProcessOtherStuff;  
    faxFinished : Finished := True;  
  end;  
until Finished;
```

Multiplexes fax transmission with other tasks (ProcessOtherStuff). When using objects, the F parameters and the CAS suffixes would be omitted.

See Also

FaxTransmit

PrepareFaxTransmitPart

Syntax

```
procedure FindFirstCAS(FP : FaxRecPtr; Q : QueueType; Direction : Boolean;  
                      Status : Integer; var Handle : Word);  
procedure CASFax.FindFirstCAS(Q : QueueType; Direction : Boolean;  
                              Status : Integer; var Handle : Word);
```

Purpose

Find the first matching event in the specified queue.

Description

This routine is a shell around CAS function \$05. It finds the first entry that matches Status in queue Q.

Entries in a queue are always in chronological order. When Direction is False, the search starts with the earliest entry and moves toward later entries. When Direction is True, the search starts with the last entry and moves toward earlier entries.

Status further qualifies the event you are seeking:

- 0 - successfully completed events
- 1 - waiting events
- 2 - event in progress
- 3 - sending event
- 4 - receiving event
- 5 - aborted events
- 1 - all events

If FindFirstCAS is successful, it sets AsyncStatus to ecOk and returns the Handle of the first event that matches your search criteria; otherwise AsyncStatus contains a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
FindFirstCAS(F, qReceive, False, 0, Handle);  
while AsyncStatus = ecOk do begin  
  MoveReceivedFile(F, Handle, 1, NextFaxName);  
  DeleteFile(F, Handle, 1, qReceive);  
  FindNextCAS(F, qReceive, Handle);  
end;
```

Finds all successfully received files in the receive queue, moves them to DOS files, and deletes the receive queue entries. FaxReceive uses logic like this to process the receive queue. When using objects, the CAS suffixes and F parameters would be omitted.

Syntax

```
procedure FindNextCAS(FP : FaxRecPtr; Q : QueueType; var Handle : Word);  
procedure CASFax.FindNextCAS(Q : QueueType; var Handle : Word);
```

Purpose

Find the next matching event in the specified queue.

Description

This routine is a shell around CAS function \$06. It finds the next entry that matches Status (set by the call to FindFirstCAS) in queue Q. You must call FindFirstCAS once before calling FindNextCAS.

If FindNextCAS is successful, it sets AsyncStatus to ecOk and returns the Handle of the next event that matches your search criteria; otherwise AsyncStatus contains a CAS error code. See the list of CAS error codes earlier in this section.

Example

See the example for FindFirstCAS.

GetAllStatus

Syntax

```
procedure GetAllStatus(FP : FaxRecPtr; var Event : Byte; var AStatus : Word;  
    var Pages : Word; var PageTotal : Word;  
    var Bytes : LongInt; var FName : PathStr;  
    var Remote : String);  
  
procedure CASFax.GetAllStatus(var Event : Byte; var AStatus : Word;  
    var Pages : Word; var PageTotal : Word;  
    var Bytes : LongInt; var FName : PathStr;  
    var Remote : String);
```

Purpose

Return selected status fields for the current event.

Description

This routine is intended primarily for internal use in FaxTransmit and FaxReceive; however, you may find it useful in your program if you're not calling FaxTransmit or FaxReceive.

All returned information pertains to the current CAS event. Event is the event number. AStatus has the following meanings:

- 0 - successfully completed
- 1 - waiting to be processed
- 2 - event in progress
- 3 - sending
- 4 - receiving
- 5 - aborted

Pages is the number of pages sent or received so far. Unfortunately, at least one version of the CAS driver always returns this value as zero. PageTotal is the total number of pages in the document; it is always zero for receive events. Bytes is the total number of bytes, for all pages, sent or received so far. FName is the name of the file being transmitted. It's an empty string for receive events. Remote is the station ID of the remote fax device.

If GetAllStatus is successful, AsyncStatus is set to ecOk, otherwise to a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
GetAllStatus(F, Event, AStatus, Pages, PageTotal, Bytes, FName, Remote);  
if AsyncStatus = ecOK then  
    WriteLn('Total bytes: ', Bytes);
```

Makes a status call to return information about the current event, then displays just the total bytes transferred so far. When using objects, the F parameter would be omitted.

Syntax

```
procedure GetEventDate(FP : FaxRecPtr; Handle : Word; Q : QueueType;  
                      var Year, Month, Day : Word);  
  
procedure CASFax.GetEventDate(Handle : Word; Q : QueueType;  
                              var Year, Month, Day : Word);
```

Purpose

Get the date for the specified event.

Description

This routine is a shell around CAS function \$0A. It returns the date associated with the event handle specified by Handle. Q is the queue to search. If the search is successful, AsyncStatus is set to ecOk and Year, Month and Day return the event date. Otherwise AsyncStatus contains a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
GetEventDate(F, Handle, qReceive, Year, Month, Day);  
if AsyncStatus = ecOk then  
  WriteLn('Fax was received on ', Month, '/', Day, '/', Year);
```

Assuming that Handle has already been set, displays the date that a fax was received.

See Also

GetEventTime

Syntax

```
procedure GetEventStatus(FP : FaxRecPtr; var SRec : StatusRecord);  
procedure CASFax.GetEventStatus(var SRec : StatusRecord);
```

Purpose

Return the status of the current event.

Description

This is a shell around CAS function \$10. It returns status information about the current event in a 512 byte structure. See StatusRecord in "Shared Declarations" for more information.

GetAllStatus may be more convenient to use than GetEventStatus, since GetAllStatus returns the same information that is available to Class 1 and Class 2 modems.

If GetEventStatus is successful, AsyncStatus is set to ecOk and the StatusRecord is filled in appropriately, otherwise AsyncStatus is set to a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
var  
  SR : StatusRecord;  
...  
  GetEventStatus(F, SR);  
  if AsyncStatus = ecOk then  
    with SR do begin  
      ...  
    end;
```

Returns status information about the current event in SR. When using objects, the F parameter would be omitted.

See Also

GetAllStatus

GetEventTime / GetExternalDataBlock

Syntax

```
procedure GetEventTime(FP : FaxRecPtr; Handle : Word; Q : QueueType;
    var Hour, Min, Sec : Word);
procedure CASFax.GetEventTime(Handle : Word; Q : QueueType;
    var Hour, Min, Sec : Word);
```

Purpose

Get the time for the specified event.

Description

This routine is a shell around CAS function \$0C. It returns the time associated with the event handle specified by Handle. Q is the queue to search. If the search is successful, AsyncStatus is set to ecOk and Hour, Min and Sec return the event time. Otherwise AsyncStatus contains a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
GetEventTime(F, Handle, qReceive, Hour, Min, Sec);
if AsyncStatus = ecOk then
    WriteLn('Fax was received at ', Hour, ':', Min);
```

Assuming that Handle has already been set, displays the time that a fax was received.

See Also

GetEventData

Syntax

```
procedure GetExternalDataBlock(FP : FaxRecPtr; var EDB : ExternalDataBlock);
procedure CASFax.GetExternalDataBlock(var EDB : ExternalDataBlock);
```

Purpose

Return information about the resident CAS driver.

Description

This routine is a shell around CAS function \$0E. It returns a 256 byte data block that contains information about the resident CAS manager. See "Shared Declarations" for more information about the ExternalDataBlock type.

If GetExternalDataBlock is successful, AsyncStatus is set to ecOk, otherwise it is set to a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
var
    EDB : ExternalDataBlock;
...
GetExternalDataBlock(F, EDB);
if AsyncStatus = ecOk then
    with EDB do begin
        ...
    end;
```

Returns information about the CAS driver in EDB. When using objects, the F parameter would be omitted.

Syntax

```
procedure GetHardwareStatus(FP : FaxRecPtr; var SArray : StatusArray);  
procedure CASFax.GetHardwareStatus(var SArray : StatusArray);
```

Purpose

Return vendor-specific hardware status information.

Description

This routine is a shell around CAS function \$12. It returns 128 bytes of vendor-specific information in a character array type called StatusArray. You'll need to refer to the manual for your CAS faxmodem to interpret this data.

If GetHardwareStatus is successful, AsyncStatus is set to ecOk, otherwise it is set to a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
var  
  S : StatusArray;  
...  
  GetHardwareStatus(F, S);  
  if AsyncStatus = ecOk then  
    {...display vendor info...}
```

Obtains a hardware status block. When using objects, the F parameter would be omitted.

Syntax

```
procedure GetPageInfoCAS(FP : FaxRecPtr; var Pages : Word; var Page : Word;  
  var BytesTransferred : LongInt;  
  var PageLength : LongInt);  
  
procedure GetPageInfo(var Pages : Word; var Page : Word;  
  var BytesTransferred : LongInt;  
  var PageLength : LongInt); virtual;
```

Purpose

Return status information about the current page.

Description

Call this routine in a fax status procedure to return information about the page currently being sent or received.

When a program is sending faxes, Pages contains the total number of pages in the current fax, Page is the number of the current page being transmitted. BytesTransferred is the number of bytes transmitted so far for this fax. PageLength is always zero, since the CAS driver does not provide this information.

When a program is receiving faxes, Pages is always zero (the total number of pages is not known in advance). The other values are interpreted the same as when transmitting CAS faxes.

Example

```
GetPageInfoCAS(F, Pages, Page, Bytes, PageLength);  
WriteLn('Bytes transferred: ', Bytes);
```

Part of a fax status procedure that displays the total number of bytes sent or received for the current fax. When using objects, the F parameter and the CAS suffix would be omitted.

GetQueueStatus / GetSetAutoReceive

Syntax

```
procedure GetQueueStatus(FP : FaxRecPtr; Q : QueueType;  
                        var Changes, ControlFiles, ReceivedFiles : Word);  
  
procedure CASFax.GetQueueStatus(Q : QueueType;  
                               var Changes, ControlFiles, ReceivedFiles : Word);
```

Purpose

Get the status of the specified queue.

Description

This routine is a shell around CAS function \$11. It returns status information for queue Q. Changes is the total number of changes made to the queue since the CAS driver was loaded. ControlFiles is the number of control files currently in the queue. ReceivedFiles, which applies only to the receive queue, is the number of files received since the CAS driver was loaded.

Example

```
GetQueueStatus(F, qTask, Changes, ControlFiles, ReceivedFiles);  
WriteLn('There are ', ControlFiles, ' files waiting to be transmitted');
```

Displays the number of faxes waiting to be transmitted. When using objects, the F parameter would be omitted.

Syntax

```
procedure GetSetAutoReceive(FP : FaxRecPtr; var Rings : Word; GetSet : Word);  
procedure CASFax.GetSetAutoReceive(var Rings : Word; GetSet : Word);
```

Purpose

Get or set the state of auto-receive.

Description

This routine is a shell around CAS function \$0F. If GetSet is zero, the current state of auto-receive is returned in Rings. If GetSet is one, Rings determines whether auto-receive is on or off. When Rings is zero, auto-receive is disabled; otherwise it is enabled to answer on the specified number of rings.

If GetSetAutoReceive is successful, AsyncStatus is set to ecOk, otherwise AsyncStatus contains a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
Rings := 1;  
GetSetAutoReceive(F, Rings, 1);
```

Enables auto-receiving by the CAS manager, which will answer incoming calls on the first ring.

See Also

FaxReceive

Syntax

constructor CASFax.Init(ID : PathStr);

Purpose

Initialize a CASFax object.

Description

This constructor instantiates an object for sending or receiving faxes through a CAS driver. It first checks to see if the CAS driver is loaded. If the driver is not loaded, the constructor fails and AsyncStatus is set to ecNoCASManager.

Example

```
var
  F : CASFaxPtr;
begin
  New(F, Init('STATION ID'));
  if F = nil then
    {...handle error...}
  ...
  Dispose(F, Done);
end.
```

Instantiates a CASFax object at the beginning of the program, destroys it at the end.

See Also

Done

Syntax

procedure InitCASFax(var FP : FaxRecPtr; ID : Str20);

Purpose

Allocate and initialize a CAS fax record.

Description

This routine first checks to see if the CAS driver is loaded. If the driver is not loaded, the routine exits immediately with AsyncStatus set to ecNoCASManager and FP set to nil. Otherwise, it allocates and initializes the CAS fax record.

Example

```
var
  F : FaxRecPtr;
begin
  InitCASFax(F, 'STATION ID');
  if F = nil then
    {...handle error...}
  ...
  DoneCASFax(F);
end.
```

Allocates a CASFax record at the beginning of the program, disposes of it at the end.

See Also

DoneCASFax

MoveReceivedFile / OpenFile

Syntax

```
procedure MoveReceivedFile(FP : FaxRecPtr; Handle : Word; FileNumber : Word;
                          NewName : PathStr);
procedure CASFax.MoveReceivedFile(Handle : Word; FileNumber : Word;
                                  NewName : PathStr);
```

Purpose

Extract a file from the receive queue to a DOS file.

Description

This routine is a shell around CAS function \$14. It extract a fax file from the receive queue and creates a DOS file. Handle is the event handle associated with the received file. FileNumber is the file number within that handle (one is the first fax file). NewName is the path and file name for the extracted file.

The file still exists in the receive queue even after it has been moved. You must specifically delete it with DeleteFile if the move is successful.

Example

See the example for FindFirstCAS.

Syntax

```
procedure OpenFile(FP : FaxRecPtr; Q : QueueType; FileNumber : Word;
                  Handle : Word; var FHandle : Word);
procedure CASFax.OpenFile(Q : QueueType; FileNumber : Word;
                          Handle : Word; var FHandle : Word);
```

Purpose

Open a file in the specified queue.

Description

This routine is a shell around CAS function \$07. Q is the queue containing the file to open. Handle is the event handle associated with the file. FileNumber is the file number within that handle (zero is the control file, one is the first fax file, etc.).

If OpenFile is successful, AsyncStatus is set to ecOk and FHandle contains the DOS file handle for the file. Otherwise AsyncStatus is set to a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
FindFirstCAS(F, qReceive, False, 0, Handle);
if AsyncStatus = ecOk then begin
  OpenFile(F, qReceive, 0, Handle, FileHandle);
  if AsyncStatus = ecOk then
    {...use FileHandle...}
end;
```

Gets an event handle for the first successfully received file in the receive queue. Then uses the event handle to open the associated control file and get a DOS file handle. DOS file read routines can then be used to read the control file.

PrepareFaxReceivePart / PrepareFaxTransmitPart

Syntax

```
procedure PrepareFaxReceivePartCAS(FP : FaxRecPtr);  
procedure CASFax.PrepareFaxReceivePart;
```

Purpose

Prepare to call FaxReceivePart.

Description

Calling this routine is necessary only if your program uses FaxReceivePart instead of FaxReceive. PrepareFaxReceivePart prepares FaxReceivePart for the initial call from your program.

Example

See FaxReceivePart for an example.

Syntax

```
procedure PrepareFaxTransmitPartCAS(FP : FaxRecPtr);  
procedure CASFax.PrepareFaxTransmitPart;
```

Purpose

Prepare to call FaxTransmitPart.

Description

Calling this routine is necessary only if your program uses FaxTransmitPart instead of FaxTransmit. PrepareFaxTransmitPart prepares FaxTransmitPart for the initial call from your program.

Example

See FaxTransmitPart for an example.

RunDiagnostics

Syntax

```
procedure RunDiagnostics(FP : FaxRecPtr; Mode : Word);  
procedure CASFax.RunDiagnostics(Mode : Word);
```

Purpose

Run CAS hardware diagnostics.

Description

This routine is a shell around CAS function \$13. If Mode is one, diagnostics are started; this request is submitted to the CAS driver and control returns to your program. If Mode is zero, AsyncStatus returns the result reported by the CAS function. The diagnostics and the values returned in AsyncStatus by the diagnostics are specific to each manufacturer. Consult your modem documentation for details. The only manufacturer-independent value is \$40, meaning that diagnostics are in progress.

Example

```
RunDiagnostics(1);  
Delay(5000);  
RunDiagnostics(0);  
WriteLn(AsyncStatus);
```

Starts hardware diagnostics and reports the results.

Syntax

```
procedure SetCASAbortFunc(FP : FaxRecPtr; CAF : AbortFunc);  
procedure CASFax.SetCASAbortFunc(CAF : AbortFunc);
```

Purpose

Set a CAS user abort function.

Description

Call this routine to specify a user-defined abort function that is active while CAS FaxReceive, FaxTransmit, FaxReceivePart, and FaxTransmitPart are running. Unlike Class 1 and Class 2 fax objects and records, CASFax cannot use the port abort function of a com port, because the CAS driver is in complete control of the com port.

The abort function type used by CAS is same as the abort function type used by com ports. In fact, com ports and CAS objects can use the same abort function.

For more information see "The CAS User Abort Hook" earlier in this section and "User Abort Function" in §4.B.

Example

```
{ $F+ }  
function KeyboardAbort : Boolean;  
begin  
  KeyboardAbort := False;  
  if KeyPressed then  
    KeyboardAbort := (ReadKey = cEsc);  
end;  
{ $F- }  
...  
SetCASAbortFunc(F, KeyboardAbort);
```

Sets an abort function that aborts FaxTransmit and FaxReceive if the user presses the <Esc> key. When using objects, the F parameter would be omitted.

See Also

FaxReceive

FaxTransmit

Syntax

```
procedure SetCASResolution(FP : FaxRecPtr; High : Boolean);  
procedure CASFax.SetCASResolution(High : Boolean);
```

Purpose

Set the fax resolution.

Description

Unlike Class 1 and Class 2 faxing, where the fax resolution is set when the document is converted to fax format, the CAS driver converts the document to fax format only after the transmission request is submitted. Pass True for High to use high resolution (200 pixels per vertical inch) or False for standard resolution (100 pixels per vertical inch).

The default resolution is set when the CAS driver is configured. Call SetCASResolution only if you don't want to use the default resolution or you don't know what the default resolution is.

Example

```
SetCASResolution(F, True);
```

All documents sent by FaxTransmit are now sent in high resolution mode. When using objects, the F parameter would be omitted.

SetLogoFile

Syntax

```
procedure SetLogoFile(FP : FaxRecPtr; LF : PathStr);  
procedure CASFax.SetLogoFile(LF : PathStr);
```

Purpose

Set a PCX logo file for the cover page.

Description

SetLogoFile specifies a PCX logo file that FaxTransmit and SubmitSingleFile will use as part of the standard format cover page when transmitting faxes. The logo file is used only if the afCASSubmitUseControl option is enabled.

LF must be the complete path name of the PCX file. The PCX file must not be larger than 1712 pixels wide by 800 pixels long (about 8.5 by 4 inches).

Example

```
SetLogoFile(F, 'J:\ASync\TPS.PCX');  
AddFaxEntry(F, '260-7151', 'J:\ASync\STUFF.TXT', 'J:\ASync\COVER.TXT');  
FaxTransmitCAS(F);
```

Sets the logo file and transmits the file STUFF.TXT. The transmitted fax will have a cover file using the PCX file TPS.PCX. When using objects, the F parameters would be omitted.

```
SetLogoFile(F, 'J:\ASync\TPS.PCX');  
AddFaxEntry(F, '260-7151', 'J:\ASync\STUFF.TXT', '');  
FaxTransmitCAS(F);
```

Incorrect. A PCX logo file was requested but the fax to be transmitted doesn't have a cover text file. FaxTransmit uses the cover text file as the signal that a cover file is to be included. If no cover text file is supplied, no cover page is transmitted. If you want a cover page without text, you must reference an empty cover text file.

Syntax

```
procedure SetTaskDate(FP : FaxRecPtr; Handle : Word; Year, Month, Day : Word);  
procedure CASFax.SetTaskDate(Handle : Word; Year, Month, Day : Word);
```

Purpose

Set the date for the specified event.

Description

This routine is a shell around CAS function \$0B. It sets the date in the control file for the task queue event associated with Handle. This is usually used to schedule fax transmit events for a specific Year, Month, and Day. If the new date is earlier than today's date, the event is executed immediately.

The range for Year is 1980-2099. The range for Month is 1-12 and for Day is 1-31.

If SetTaskDate is successful, AsyncStatus is set to ecOk, otherwise it is a CAS error code. See the list of CAS error codes earlier in this section.

Note that FaxTransmit does not use SetTaskDate or SetTaskTime. Faxes sent using FaxTransmit are always scheduled for immediate transmission.

Example

```
SetTaskDate(F, Handle, 1994, 1, 1);
```

Sets the task date for Handle to January 1, 1994. When using objects, the F parameter would be omitted.

See Also

GetEventData

SetTaskTime

Syntax

```
procedure SetTaskTime(FP : FaxRecPtr; Handle : Word; Hour, Min, Sec : Word);  
procedure CASFax.SetTaskTime(Handle : Word; Hour, Min, Sec : Word);
```

Purpose

Set the time for the specified event.

Description

This routine is a shell around CAS function \$0D. It sets the time in the control file for the task queue event associated with Handle. This is usually used to schedule fax transmit events for a specific Hour, Min, and Sec. If the new date and time is earlier than the current date and time, the event is executed immediately.

The range for Hour is 0-23. The range for Min and Sec is 0-59.

If SetTaskTime is successful, AsyncStatus is set to ecOk, otherwise it is a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
SetTaskTime(F, Handle, 0, 0, 0);
```

Sets the task time for Handle to midnight. When using objects, the F parameter would be omitted.

See Also

GetEventTime

SetTaskDate

SubmitSingleFile

Syntax

```
procedure SubmitSingleFile(FP : FaxRecPtr; TransType : Word; TextSize : Word;  
                           Time : Word; Date : Word; Dest : String;  
                           FName : PathStr; Number : String;  
                           Cover : PathStr; var Handle : Word);  
  
procedure CASFax.SubmitSingleFile(TransType : Word; TextSize : Word;  
                                  Time : Word; Date : Word; Dest : String;  
                                  FName : PathStr; Number : String;  
                                  Cover : PathStr; var Handle : Word);
```

Purpose

Submit a single fax transmission to the CAS manager.

Description

This routine is a shell around CAS function \$15. It is a convenient method of scheduling the fax transmission of a single text file. The alternative is to create and fill in a control file, then submit the control file using SubmitTask.

TransType is the requested resolution; zero for high resolution, one for standard resolution. TextSize refers the number of columns accepted in text documents: zero for 80 columns, one for 132 columns. Time and Date are the time and date to send the fax; zero values mean the fax should be sent immediately. If non-zero, Time and Date should be in DOS packed directory format.

Dest is the name of the recipient. It is included in the header at the top of each transmitted page. FName is the name of the text file to transmit. It *must* include path information unless the file is in the outbox directory specified when you configured the CAS driver.

Number is the phone number of the remote fax device. Cover is the name of an optional text file containing cover page text, including replacement tags if desired. This file must be no larger than MaxCoverData (2048) bytes in size, after any substitutions are made.

If the file is successfully submitted, AsyncStatus is set to ecOk and Handle contains the event handle assigned by CAS, otherwise AsyncStatus contains a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
SubmitSingleFile(F,  
                 0,                               {High resolution}  
                 0,                               {80 columns}  
                 0,                               {Time = now}  
                 0,                               {Date = now}  
                 '',                               {No recipient name}  
                 'J:\OUT\STUFF.TXT',              {File to send}  
                 '1-719-260-7151',                {Number to dial}  
                 '',                               {No cover sheet}  
                 Handle);                         {Returned event handle}  
  
if AsyncStatus = ecOk then  
    WriteLn('Fax submitted, handle is ', Handle);
```

Submits the text file J:\OUT\STUFF.TXT for transmission as a high resolution fax to 719-260-7151. When using objects, the F parameter would be omitted.

See Also

SubmitTask

Syntax

```
procedure SubmitTask(FP : FaxRecPtr; FName : PathStr; var Handle : Word);  
procedure CASFax.SubmitTask(FName : PathStr; var Handle : Word);
```

Purpose

Submit a task to the CAS manager.

Description

This is a shell around CAS function \$01. It submits a preformatted control file to the task queue. FName is the complete path and file name of a control file that you have built. You will need to refer to the CAS specification to properly build the control file. See the CAS specification stored in the self-extracting archive CASxx.COM in your \APRO\BONUS subdirectory.

If SubmitTask is successful, AsyncStatus is set to ecOk and Handle contains the event handle, otherwise AsyncStatus contains a CAS error code. See the list of CAS error codes earlier in this section.

Example

```
BuildControlFile(...);  
SubmitTask(F, 'J:\OUT\STUFF.CTL', Handle);
```

BuildControlFile builds an appropriate control file and SubmitTask submits it to the CAS manager.

See Also

SubmitSingleFile

G. Demonstration Programs

1. Simple Send Fax Program: SIMPSND/SIMPSNDO

These two programs are simple demonstrations of how to send a fax using Class 1, Class 2, or CAS services. The procedural version (SIMPSND) and the object-oriented version (SIMPSNDO) are used in exactly the same way. Each expects a command line of the following form:

```
SIMPSND [options] FaxFile
```

FaxFile is the name of the fax file to send. A file extension of APF is assumed if none is specified. The filename cannot include wildcard characters—only one fax file can be sent at a time.

The following options can be specified:

```
/I StationID  Set station ID
/N Number     Set fax phone number to call
/C ComPort    Set com port (1-4, default=1)
/B Baud       Set port baud rate (4800, 7200, 9600, 19200; default=19200)
/F FaxBPS     Set fax BPS rate (2400, 4800, 7200, 9600, 12000, 14400;
               default = 9600)
/M InitString Set modem initialization string
/V CoverFile  Set cover page file
/H           Use highest possible bit rate, and ECM if available
/S FaxClass   Set fax class (1, 2, A(uto), C(AS); default = 2)
/?           Display help screen
```

Note that '-' can be used instead of '/' when specifying command line options.

The station ID set by the /I option can contain up to 20 characters and is used to identify the sending fax station. The Class 1 and Class 2 specifications indicate that the station ID should contain just your telephone number and therefore limit it to just the digits 0 through 9 and space. However, SIMPSND doesn't filter the characters you specify, and most faxmodems will accept all characters. Note, however, that the first space in StationID terminates the string. If you need to use embedded spaces, enclose the station ID in double quotes. If you don't set the Station ID, it is blank, which is acceptable, but the receiver will not receive an identification of your station.

The /N option allows you to specify the phone number to call. Number can contain any 40 characters, but all of them will be sent to the modem, so you should not include any characters that it will not understand within a phone number. Number cannot contain embedded spaces, because a space is used as the end delimiter of the phone number.

/B sets the baud rate between the PC and the modem.

/F sets the connection speed between the two modems. This is the initial BPS rate that the local modem will try to achieve. If the remote modem doesn't support this speed, it will be negotiated down. Warning: no check is made to ensure that the BPS rate you specify here is supported by your modem. If it is not supported, SIMPSND may fail during the initialization step.

Note that Baud (/B) must always be *higher* than FaxBPS (/F) because the faxmodem requires a continuous stream of data once the transmission is started. This can only be guaranteed if Baud is higher than FaxBPS.

The /M option specifies a string that is sent during modem initialization. It allows you to specify modem preferences (see SetModemInit in APFAX12/OOFAX12).

Use the /V option to specify a cover page file. See "Cover Pages" in §10.D for more information.

When /H is specified, SIMPSND interrogates the modem to find the highest BPS it supports and whether Error Correcting Mode (ECM) is available. These options are then set based on the modem response. Warning: a few modems just say "yes" to everything, so you must be sure that your modem can respond appropriately before using this option.

The /S option allows you to specify the fax class. The valid classes are Class 1, Class 2, and CAS. If you specify A, SIMPSND automatically determines the fax class for you.

2. Simple Receive Fax Program: SIMPRCV/SIMPRCVO

These two programs are simple demonstrations of how to receive a fax using Class 1, Class 2, or CAS services. The procedural version (SIMPRCV) and the object-oriented version (SIMPRCVO) are used in exactly the same way. Each expects a command line of the following form:

SIMPRCV ComPort [options]

ComPort is the number of the com port to be used. This parameter must always be specified. Valid values are 1 through 4. If you specify a CAS receive using the '/S C' option, described below, no ComPort is necessary and its value is ignored if you specify one.

The following options can be specified:

/I StationID	Set station ID
/B Baud	Set port baud rate (4800, 7200, 9600, 19200; default=19200)
/F FaxBPS	Set fax BPS rate (2400, 4800, 7200, 9600, 12000, 14400; default = 9600)
/M InitString	Set modem initialization string
/S FaxClass	Set fax class (1, 2, C(AS), A(uto); default = 2)
/?	Display help screen

Note that '-' can be used instead of '/' when specifying command line options.

The station ID set by the /I option can contain up to 20 characters and is used to identify the receiving fax station. The Class 1 and Class 2 specifications indicate that the station ID should contain just your telephone number and therefore limit it to just the digits 0 through 9 and space. However, SIMPRCV doesn't filter the characters you specify, and most faxmodems will accept all characters. Note, however, that the first space in StationID terminates the string. If you need to use embedded spaces, enclose the station ID in double quotes. If you don't set the station ID, it is blank, which is acceptable, but the sender will not receive an identification of your station.

/B sets the baud rate between the PC and the modem.

/F sets the connection speed between the two modems. This is the initial BPS rate that the local modem will try to achieve. If the remote modem doesn't support this speed, it will be negotiated down. Warning: no check is made to ensure that the BPS rate you specify here is supported by your modem. If it is not supported, SIMPRCV may fail during the initialization step.

Note that Baud (/B) must always be *higher* than FaxBPS (/F) because the faxmodem expects the receiving terminal to be able to accept incoming data as fast as the faxmodem can provide it.

The /M option specifies a string that is sent during modem initialization. It allows you to specify modem preferences (see SetModemInit in APFAX12/OOFAX12).

The /S option allows you to specify the fax class. The valid classes are Class 1, Class 2, and CAS. If you specify A, SIMPRCV automatically determines the fax class from information provided by the modem.

3. Convert to Fax: CVT2FAX/ CVT2FAXO

These two programs demonstrate how to convert text, PCX, and TIFF files to Async Professional fax (APF) format. The procedural version (CVT2FAX) and the object-oriented version (CVT2FAXO) are used in exactly the same way. Each expects a command line of the following form:

```
CVT2FAX [options] FileName
```

FileName is the name of the text, PCX, or TIFF file to convert. An extension of TXT, PCX, or TIF is assumed, respectively, if none is provided. The filename cannot contain wildcard characters.

The following options can be specified:

/T	Convert from ASCII text to fax (default)
/P	Convert from PCX (mono) to fax
/F	Convert from TIFF (mono) to fax
/L Lines	Specify page break (text only, default=0 (no breaks))
/H	High resolution output (200x200; default=200x100)
/B	Use high resolution font when converting text
/S FontName	Use HP soft font file
/N	No width doubling for graphics files
/J	Left justify graphics images instead of centering them
/?	Display help screen

Note that '-' can be used instead of '/' when specifying command line options.

The /T, /P and /F parameters specify the type of file to convert: ASCII text file, PCX file or TIFF graphics files.

The /L parameter selects the number of lines per page in a text conversion. A zero value indicates no page breaks are used.

The /H parameter specifies high resolution (200x200 dots per inch) output. The default is standard resolution (200x100 dots per inch).

The /B parameter specifies high resolution font. It should be used when converting text files using high resolution (/H).

The /S parameter specifies an H-P font file name. The H-P font file must follow the rules for H-P font files described in the reference for LoadHPFont.

The width of all graphics images is doubled in standard resolution to keep the proper aspect ratio. Use the /N parameter to disable width doubling.

Graphics images are centered in the middle of the fax page. Use the /J parameter to disable this behavior, graphics images will be left justified on the fax page.

4. Fax File Display: SHOWFAX/SHOWFAXO

These two programs are simple demonstrations of how to display an Async Professional (APF format) fax file. To keep the programs simple, the display methodology used is also quite simple. Only a VGA adapter that supports BIOS video mode \$12 (640x480 monochrome) is supported.

The procedural version (SHOWFAX) and the object-oriented version (SHOWFAXO) are used in exactly the same way. Each expects a command line of the following form:

```
SHOWFAX [options] FaxFile
```

FaxFile is the name of the fax file that will be displayed. A file extension of APF is assumed if none is specified. The filename cannot include wildcard characters—only one fax file can be displayed at a time.

The following options can be specified:

/D	Reduce image width (50% reduction)
/P Page	Starting page number
/R	Remove display after 2 seconds
/?	Display help screen

Note that '-' can be used instead of '/' when specifying command line options.

The /D parameter indicates that a reduced (50%) image should be displayed.

The /P parameter allows you to specify the first page to be displayed. Note that the entire file is unpacked for display, so you can scroll to any page in the file after the requested page is displayed.

The /R parameter specifies that the display should automatically be removed from the screen after 2 seconds. The prompt before the fax is displayed is also eliminated. This is primarily for internal use in automated testing.

While viewing the fax file, you can use the following keys to move within the file:

<Down>

Move down 1 pixel.

<Up>

Move up 1 pixel.

<Right>

Move right 64 pixels.

<Left>

Move left 64 pixels.

<PgDn>

Move down 72 pixels. Note that a complete fax page cannot usually be displayed on one screen.

<PgDn> moves down 72 pixels within the current page. To go to the next fax page, use <N>.

<PgUp>

Move up 72 pixels. Note that a complete fax page cannot usually be displayed on one screen.

<PgUp> moves up 72 pixels within the current page. To go to the previous fax page, use .

<N>

Go to the beginning of the next fax page.

Go to the beginning of the previous fax page.

<L>

Go to the beginning of the last fax page.

<F>

Go to the beginning of the first fax page.

<Esc>

Exit SHOWFAX.

5. Fax File Print: PRNFAX/PRNFAXO

These two programs are simple demonstrations of how to print a fax to PCL5-compatible (e.g., H-P LaserJet) or Epson-compatible printers. The procedural version (PRNFAX) and the object-oriented version (PRNFAXO) are used in exactly the same way. Each expects a command line of the following form:

```
PRNFAX [options] FaxFile
```

FaxFile is the name of the fax file that will be printed. A file extension of APF is assumed if none is specified. The filename cannot include wildcard characters—only one fax file can be printed at a time.

The following options can be specified:

/C	Use TIFF data compression
/E	Use Epson FX format output (default = HP PCL5)
/F FileName	Write output to a file rather than a printer
/3	Use 300x300 resolution (default = 150x150)
/S	Disable page scaling
/?	Display help screen

Note that '-' can be used instead of '/' when specifying command line options.

The /C parameter allows you to choose TIFF data compression for H-P printers. Fax images can get rather large—multipage faxes can exceed one megabyte of data. Even on fast printers, it can take a long time (possibly several minutes) to print such images. The PCL5 printer language offers several options for compressing data sent to the printer with the TIFF compression mode being the most effective. This option tells PRNFAX to use TIFF compression as it sends data to the printer. TIFF compression generally reduces the amount of data to transmit by a factor of three.

The /E parameter allows you to choose Epson FX format for the output. The default format for output is H-P PCL5.

The /F parameter specifies that the output should be written to file FileName instead of to a printer.

The /3 parameter specifies that 300x300 resolution should be used on PCL5 printers instead of the default 150x150.

The fax data is scaled to the proper size for the current printer and mode by default. Use the /S option if you need to disable scaling.

6. Convert Fax to PCX: FAX2PCX/FAX2PCXO

These two programs allow you to convert Async Professional (APF format) fax files to PCX format. The procedural version (FAX2PCX) and the object-oriented version (FAX2PCXO) are used in exactly the same way. Each expects a command line of the following form:

`FAX2PCX [options] FaxFile`

FaxFile is the name of the fax file that will be converted. A file extension of APF is assumed if none is specified. The filename cannot include wildcard characters—only one fax file can be converted at a time.

The following options can be specified:

`/H` Reduce image width (50% reduction)
`/?` Display help screen

Note that '-' can be used instead of '/' when specifying command line options.

The `/F` parameter specifies that the width of the output should be reduced by 50%.

7. Background TSR Fax Receiver: RFAX/RFAXO

These two programs implement a TSR that can receive faxes while you do other work. The TSR works using Class 1 or Class 2 services only. The procedural version (RFAX) and the object-oriented version (RFAXO) are used in exactly the same way. Each expects a command line of the following form:

RFAX ComPort [options]

ComPort is the number of the com port to be used. This parameter must always be specified. Valid values are 1 through 4.

The following options can be specified:

/U	Unload the resident copy of RFAX
/R	Tell the resident copy of RFAX to reset the modem
/D	Disable (but leave loaded) the resident copy of RFAX
/T	Take full control during receives (default=full background)
/B Baud	Set port baud rate (4800, 7200, 9600, 19200; default = 19200)
/F FaxBPS	Set fax BPS rate (2400, 4800, 7200, 9600, 12000, 14400; default = 9600)
/M InitString	Set modem initialization string
/I StationID	Set station ID
/P Path	Set received fax files path
/S FaxClass	Set fax class (1, 2, A(uto); default = 2)
/?	Display help screen

Note that '-' can be used instead of '/' when specifying command line options.

The /R option allows you to reset the modem to receive fax mode. This might be useful if you are running other communication programs that use the same port.

When /T is specified, RFAX takes full control whenever faxes are being received. This means that RFAX watches for incoming calls while in the background. When a call comes in, however, RFAX pops up and remains in control to receive the entire fax before popping down and giving control back to the foreground program. In full background operation (the default), RFAX runs entirely in the background (it watches for a call and receives the fax without interrupting the foreground program). /T is useful on slow machines or when the foreground process is using DOS services too frequently for the fax routines to get sufficient time.

/B sets the baud rate between the PC and the modem.

/F sets the connection speed between the two modems. This is the initial BPS rate that the local modem will try to achieve. If the remote modem doesn't support this speed, it will be negotiated down. Warning: no check is made to ensure that the BPS rate you specify here is supported by your modem. If it is not supported, RFAX may fail during the initialization step.

Note that Baud (/B) must always be *higher* than FaxBPS (/F) because the faxmodem expects the receiving terminal to be able to accept incoming data as fast as the faxmodem can provide it.

The /M option specifies a string that is sent during modem initialization. It allows you to specify modem preferences (see SetModemInit in APFAX12/OOFAX12).

The station ID set by the /I option can contain up to 20 characters and is used to identify the receiving fax station. The Class 1 and Class 2 specifications indicate that the station ID should contain just your telephone number and therefore limit it to just the digits 0 through 9 and space. However, SIMPRCV doesn't filter the characters you specify, and most faxmodems will accept all

characters. Note, however, that the first space in StationID terminates the string. If you need to use embedded spaces, enclose the station ID in double quotes. If you don't set the station ID, it is blank, which is acceptable, but the sender will not receive an identification of your station.

Use the /P option to specify a path for storing received fax files. The default path is the current directory.

The /S option allows you to specify the fax class. The valid classes are class 1 and class 2. If you specify A, RFAX automatically determines the fax class from the information sent by the modem.

8. Dedicated Fax Receiver/Printer: FAXSRVR/FAXSRVRO

These two programs provide a foreground fax server that receives, logs, and prints incoming faxes on PCL5-compatible (e.g., H-P LaserJet) or Epson-compatible printers. The fax receiver supports Class 1 and Class 2 modems only. The procedural version (FAXSRVR) and the object-oriented version (FAXSRVRO) are used in exactly the same way. Each expects a command line of the following form:

FAXSRVR ComPort [options]

ComPort is the number of the com port to be used. This parameter must always be specified. Valid values are 1 through 4.

The following options can be specified:

/B Baud	Set port baud rate (4800, 7200, 9600, 19200; default = 19200)
/F FaxBPS	Set fax BPS rate (2400, 4800, 7200, 9600, 12000, 14400; default = 9600)
/P Dest	Set print destination (PRN, LPT1, ..., or FileName)
/T	Suspend printing of received faxes
/D Path	Fax file source/destination path
/3	Use 300x300 resolution (default = 150x150)
/E	Use Epson FX format output (default = H-P PCL5)
/I StationID	Set station ID
/S	Disable page scaling
/?	Display help screen

Note that '-' can be used instead of '/' when specifying command line options.

/B sets the baud rate between the PC and the modem.

/F sets the connection speed between the two modems. This is the initial BPS rate that the local modem will try to achieve. If the remote modem doesn't support this speed, it will be negotiated down. Warning: no check is made to ensure that the BPS rate you specify here is supported by your modem. If it is not supported, FAXSRVR may fail during the initialization step.

Note that Baud (/B) must always be *higher* than FaxBPS (/F) because the faxmodem expects the receiving terminal to be able to accept incoming data as fast as the faxmodem can provide it.

Use the /P option to specify the print destination (a printer or file).

The /T option disables printing of fax files until the program is either restarted or the "Toggle Printing" option is selected. (See the help screen section below.)

The /D option sets the path for storing received fax files and getting fax files to print.

The /3 parameter specifies that 300x300 resolution should be used on PCL5 printers instead of the default 150x150.

The /E option allows you to choose Epson FX format for the output. The default format for output is H-P PCL5.

The station ID set by the /I option can contain up to 20 characters and is used to identify the receiving fax station. The Class 1 and Class 2 specifications indicate that the station ID should contain just your telephone number and therefore limit it to just the digits 0 through 9 and space. However, SIMPRCV doesn't filter the characters you specify, and most faxmodems will accept all characters. Note, however, that the first space in StationID terminates the string. If you need to use

embedded spaces, enclose the station ID in double quotes. If you don't set the station ID, it is blank, which is acceptable, but the sender will not receive an identification of your station.

The fax data is scaled to the proper size for the current printer and mode by default. Use the /S option if you need to disable scaling.

After executing the program, there are several initializing steps displayed on the screen. If the initialization fails, an error message is displayed and control is returned to DOS. If initialization is successful, the main screen is displayed:

Receive-Fax Server (c) 1993 TurboPower Software	
Chip: ROCKWELL Class: Class 2 Model: RC32ACL Revision: V1.300-AS29 Station ID: TPS (719) 260-7151	
Connection speed: Remote ID: (No remote connection) Error correction: High resolution:	Faxes received: File name: Bytes received: Page:
Faxes to print: 0 Fax file path: C:\FAX	Port: LPT1 Printing: (Nothing) Page: of:
<P>urge faxes <V>iew log <D>elete log <S>hell to DOS <T>oggle printing	
Status: Waiting to receive fax Press <Esc> to exit	

The first section shows information reported by the faxmodem: Chip, Class, Model number, and Revision number. This information is not provided by some Class 1 faxmodems. In addition, the local station ID is shown.

The second section shows information that is specific to the current connection. The connection speed, remote identification (if available), the error correction mode, and the resolution mode are shown on the left. The total count of faxes received since the program was started, the local fax file name, the number of data bytes received, and the current page number of the incoming fax are displayed on the right side. If there isn't a current connection, the Remote ID field displays '(No remote connection)'.

The third section shows the total number of faxes that require printing, the path where fax files are stored, and the printer port. The Printing field displays the name of the file that is currently printing. If all fax files have been printed, '(Nothing)' is displayed instead of a file name. If printing has been suspended or if the printer is off or off-line, '(Suspended)' is displayed.

The fourth section displays the commands available in FAXSRVR:

<P>

Deletes all faxes in the fax directory that have already been printed.

<V>

Displays a window containing the contents of the fax log file.

<D>

Deletes the fax log file.

<S>

Allows you to temporarily shell to DOS. If an incoming fax is detected, the computer's speaker will sound an alarm indicating that you should exit the DOS shell by typing 'EXIT'.

<T>

Toggles fax printing. This command suspends or enables printing of received fax files. Fax printing is enabled by default unless the /S command line option is specified.

The last line is a multi-purpose area that normally displays the status of the current operation, but is also used as a prompt line when the <P>urge faxes or <D>delete log command is selected.

While a fax is being received or printed, all commands except <Esc> are disabled. Pressing <Esc> at any time aborts the current process and exits the application. The status line displays "Shutting down" during the exit process.

11. Data Compression

A. Overview

One of the more common uses of asynchronous communications routines such as those in Async Professional is to develop programs that transfer files from one machine to another, either directly or via modems. And more and more, users are compressing their files before transferring them. Although there are many utility programs available that compress files and group the compressed files together in a single file (an "archive"), two programs have distinguished themselves in the eyes of most users, PKZIP and LHARC.

PKZIP is a shareware product developed by Phil Katz of PKWARE, Inc., that quickly and efficiently compresses multiple files and stores them in a single file, called a ZIP file. Although both the file format and the name "ZIP" have been placed in the public domain, and the basic algorithm used for extracting files from a ZIP archive is documented, the actual source code for PKZIP and its companion program, PKUNZIP, are not available for public consumption. Since PKZIP and PKUNZIP are shareware products, neither program may be distributed freely.

PKZIP provides two formats for compressing files. The ZIP "shrink" format (actually, LZW compression) provides a reasonable compromise between compression efficiency and compression speed. Though it's not as efficient as the ZIP "implode" format, it's generally much faster. You would use shrink format when compression speed is more important than compression efficiency. The new ZIP 2.0 deflate/inflate format is not supported.

LHARC was developed by Haruyasu Yoshizaki (Yoshi) of Japan. Combining the functions of PKZIP and PKUNZIP, LHARC allows you to compress multiple files into a single file and later extract and decompress them. Although it is slower than PKZIP, it produces files of comparable size. Its chief advantage over PKZIP is that it may be distributed freely, making it feasible to distribute LZH files as part of a commercial product, since LHARC itself can be distributed with them.

LHA is the successor to LHARC. It too produces LZH files, although it uses a different method for freezing the files. The LHA compression method produces a more compact LZH file. The APLZH and OOLZH modules work with both LHARC version 1.x and LHA version 2.x.

The six units described in this chapter provide routines/objects that can be used to view the directories of LZH and ZIP archives, to compress files and place them in LZH and ZIP archives, and to extract files from them. Together they make it possible to develop programs for compressing and decompressing files, capabilities that are particularly useful when writing programs that transfer files between machines.

The units fall into two groups. The first group consists of APARCHIV, APLZH, and APZIP. These units are the procedural implementations of the compression utilities. The second group consists of OOARCHIV, OOLZH, and OOZIP. As you might expect, these units provide a set of objects that implement the same functions as the routines in the other three units. In accordance with the design principles of Async Professional, the methods for these objects generally have a one-to-one correspondence with routines in APARCHIV, APLZH, and APZIP, and wherever feasible the method has the same name as the corresponding procedure or function.

B. Archive Utility Functions: APARCHIV/OOARCHIV

These two units provide facilities that are useful when working with archives in both the LZH and the ZIP formats. APARCHIV offers procedures and functions for setting options that affect the handling of archives by the higher-level units APLZH and APZIP, as well as a variety of routines for working with file mask lists (a concept described below).

OOARCHIV provides the same basic functions, implemented in the form of objects. The Archive object is an abstract type from which the higher-level objects UnLzh, Lzh, UnZip, and Zip are derived. It defines the data fields and methods shared in common by its descendants. OOARCHIV also provides the FileMaskList and FileMaskNode objects. FileMaskLists are used to create and manage singly-linked lists of file masks (e.g., '*.PAS', '*.EXE'). A FileMaskNode is simply a single node within such a list.

File Mask Lists

The best way to explain the purpose of a file mask list is to consider the following example:

Suppose you are writing a program that needs to extract selected files from an LZH file. You want the user to be able to specify the files to extract in terms of DOS file masks--say, *.PAS, *.EXE, and *.DOC. To accomplish this task, you would create a file mask list and add each of the three file masks to the list. To extract all files matching any of the three masks from the archive, you would then call ExtractFileMaskList.

Here's how to do it using APARCHIV and APLZH (the name of the LZH file and the file masks are hard-coded here to simplify the example):

```
uses
  ApMisc, ApArchiv, ApLzh;
var
  FML : FileMaskList;
begin
  {create the file mask list}
  InitFileMaskList(FML);
  if not AppendFileMask('*.PAS', FML) then {handle error};
  if not AppendFileMask('*.EXE', FML) then {handle error};
  if not AppendFileMask('*.DOC', FML) then {handle error};

  {open the LZH file}
  InitLzhFile('FOO.LZH');
  if ArchiveStatus <> 0 then {handle error};

  {extract all compressed files matching one of the masks}
  ExtractFileMaskList(FML);
  if ArchiveStatus <> 0 then {handle error};

  {close the LZH file}
  DoneLzhFile;

  {dispose of the file mask list}
  DoneFileMaskList(FML);
end.
```

The object-oriented version of the program is quite similar:

```
uses
  ApMisc, OoArchiv, OoLzh;
var
  FML : FileMaskList;
  UL : UnLzh;
```



```

begin
  {create the file mask list}
  FML.Init;
  if not FML.Append('*.PAS') then {handle error};
  if not FML.Append('*.EXE') then {handle error};
  if not FML.Append('*.DOC') then {handle error};

  {open the LZH file}
  if not UL.Init('F00.LZH') then {handle error};

  {extract all compressed files matching one of the masks}
  UL.ExtractFileMaskList(FML);
  if UL.GetLastError <> 0 then {handle error};

  {close the LZH file}
  UL.Done;

  {dispose of the file mask list}
  FML.Done;
end.

```

In both versions, ExtractFileMaskList looks at each file in the archive's directory and checks to see if it matches one of the three file masks (using MatchFileMask or FileMaskList.Match). If it does, the file is extracted.

Important: An empty file mask list is considered equivalent to a mask of '*.*'. So, in the above examples, if you were to delete the calls to AppendFileMask and FileMaskList.Append, the call to ExtractFileMaskList would extract all files in the archive.

Optional Settings

APARCHIV and OOARCHIV also allow you to set options that affect the processing of LZH and ZIP files. The SetOutputPath procedure in APARCHIV and the Archive.SetOutputPath method in OOARCHIV can be used when extracting files to specify the drive:\directory where the extracted files are to be written.

The other available options are selected using the arOptionsOn procedure/method. As elsewhere in Async Professional, these options are represented as bit flags, which can be combined by adding or OR'ing them together. For example, to select both the arReadArcComments and the arReadFileComments options, use

```
arOptionsOn(arReadArcComments+arReadFileComments);
```

or

```
arOptionsOn(arReadArcComments OR arReadFileComments);
```

The arCreateDirs option pertains to both LZH and ZIP files. If it is off, as it is by default, and a file in an archive has a relative pathname (e.g. 'DOC\README.DOC'), it is written to the default path for output files (e.g. 'C:\FILES') when it is dearchived. If the option is on, it is written to the indicated directory (e.g. 'C:\FILES\DOC'), which is created if necessary.

The arReadArcComments and arReadFileComments options currently pertain only to ZIP files, since the LZH file format makes no provisions for comments. These options determine whether or not the comments stored in an archive will be read into memory when the appropriate section of the archive is being processed. arReadArcComments pertains to the comments for the archive as a whole (general comments entered by the user concerning the contents of the ZIP file). arReadFileComments pertains to the comments for individual files (specific comments concerning a particular file stored in the ZIP file).

The `arStripPath` option tells the compressor to strip the path from the supplied file names as they are stored in the archive.

The `arCompressing` and `arDeleting` options are "information only" options that you can check in your `ShowProgress` and `ShowName` hooks (or any other user hooks) so that you can display exactly what action is taking place.

`arNoDriveLetter` is used for compatibility with PKZIP 2.0. When this option is set, the zipping routines do not store the drive letter with the path. If the drive letter is stored, PKZIP 2.0 reports "invalid file name" and refuses to decompress the file. Apparently, not storing the drive letter is the proper behavior since no version of PKZIP stores them. However, the Async Professional unzipping routines and PKUNZIP 1.0 tolerate and honor the drive letter if it's specified. `arNoDriveLetter` is not set by default to maintain compatibility with earlier versions of Async Professional.

When `arRemoveDots` is set, as it is by default, relative paths are removed from file names as they are stored. For example, `..\TEMP\XXX` is stored as `TEMP\XXX`. This is true for both ZIP and LZH archives.

Error Handling

Errors that occur in `APARCHIV`, `APLZH`, or `APZIP` are reported in the `ArchiveStatus` variable declared in `APMISC`. `ArchiveStatus` is to these three units what `AsyncStatus` is to the other units in Async Professional.

Errors occurring in one of the object-oriented units (`OOARCHIV`, `OOLZH`, or `OOZIP`) are handled a bit differently. When working with an object of type `UnLzh`, `Lzh`, `UnZip`, or `Zip`, you generally check for errors by calling the `GetLastError` method, which they inherit from `Archive`. However, an error occurring in the constructor of an `UnLzh`, `Lzh`, `UnZip`, or `Zip` object is reported in `ArchiveStatus`:

```
if not UnLzh.Init('FOO.LZH') then
  { check ArchiveStatus for error code} ;
```

Shared Declarations

Constants

```
arCreateDirs      = $0001;
arReadArcComments = $0002;
arReadFileComments = $0004;
arStripPath       = $0008;
arCompressing     = $0100;
arDeleting        = $0200;
arNoDriveLetter   = $0400;
arRemoveDots      = $0800;
```

These options, which can be turned on or off with `arOptionsOn` and `arOptionsOff`, influence how archives are read and processed. If the `arCreateDirs` option is off, and a file in an archive has a relative pathname (e.g. `DOC\README.DOC`), it is written to the default path for output files (e.g. `C:\FILES`) when it is dearchived. If the option is on, it is written to the indicated directory (e.g. `C:\FILES\DOC`), which is created if necessary.

The `arReadArcComments` and `arReadFileComments` options determine whether or not the comments stored in an archive are read into memory when the appropriate section of the archive is processed. `arReadArcComments` pertains to the comments for the archive as a whole. `arReadFileComments` pertains to the comments for individual files. These options affect the handling only of ZIP files, since LZH files currently do not provide a facility for storing comments in the archive.

The `arStripPath` option tells the compressor to strip the path from the supplied file names as they are stored in the archive. The compressor still honors the path name when finding the file to compress.

The `arCompressing` and `arDeleting` options are "information only" options—you should never change these options. You can check them in your `ShowProgress` and `ShowName` hooks (or any other user hooks) so that you can display exactly what action is taking place. If neither option is set, files are currently being extracted from the archive.

`arNoDriveLetter` is used for compatibility with PKZIP 2.0. When this option is set, the zipping routines do not store the drive letter with the path. If the drive letter is stored, PKZIP 2.0 reports "invalid file name" and refuses to decompress the file. Apparently, not storing the drive letter is the proper behavior since no version of PKZIP stores them. However, the Async Professional unzipping routines and PKUNZIP 1.0 tolerate and honor the drive letter if it's specified. `arNoDriveLetter` is not set by default to maintain compatibility with earlier versions of Async Professional.

When `arRemoveDots` is set, as it is by default, relative paths are removed from file names as they are stored. For example, `..\\TEMP\\XXX` is stored as `TEMP\\XXX`. This is the same behavior used by PKZIP 1.0/2.0 and is useful for recreating directory structures from any root directory. However, it is different from the way names were stored prior to Async Professional 1.12, which stored relative path information in the archive. If you want the old behavior of Async Professional, turn off `arRemoveDots`.

`BadArchiveOptions` : Word = `arReadExtraField`;

Used to designate archive options that are intended solely for internal use, and should not be changed.

`DefArchiveOptions` : Word = `arRemoveDots`;

This constant is used to store the default archive options.

APARCHIV Declarations

Constants

`arOutPath` : PathStr = '';

This internal variable is used to store the name of an output path, as specified by a call to `SetOutputPath`.

Types

```
FileMaskList =  
  record  
    Head, Tail : FileMaskNodePtr;  
  end;
```

A singly-linked list of file masks. Head points to the first node in the list, Tail to the last node.

```
FileMaskNodePtr = ^FileMaskNode;  
FileMaskNode =  
  record  
    DirPtr : StringPtr;  
    Name : NameStr;  
    Ext : ExtStr;  
    Next : FileMaskNodePtr;  
  end;
```

A node in a list of file masks. Name and Ext store the actual file mask (e.g., '*' and 'DOC' for '*.DOC'). DirPtr stores the path information from the PathStr passed to `Append` or `AppendFileMask`. Next points to the next file mask in the list.

OOARCHIV Declarations

Types

```
ArchivePtr = ^Archive;
Archive =
  object
    arError   : Word;
    arFile    : File;
    arName    : PathStr;
    arOutPath : PathStr;
    arOptions : Word;
    ...
  end;
```

An abstract object. Descendants of Archive are used for extracting information and/or files from an archive (i.e., a ZIP or LZH file).

```
FileMaskListPtr = ^FileMaskList;
FileMaskList =
  object
    fmlHead, fmlTail : FileMaskNodePtr;
    ...
  end;
```

A singly-linked list of file masks. fmlHead points to the first node in the list, fmlTail to the last node.

```
FileMaskNodePtr = ^FileMaskNode;
FileMaskNode =
  object
    fmnNext : FileMaskNodePtr;
    fmnDirPtr : StringPtr;
    fmnName : NameStr;
    fmnExt : ExtStr;
    ...
  end;
```

A node in a list of file masks. fmnName and fmnExt store the actual file mask (e.g., '*' and 'DOC' for '*.DOC'). fmnDirPtr stores the path information from the PathStr passed to Append or AppendFileMask. fmnNext points to the next file mask in the list.

Syntax

```
function FileMaskList.Append(FM : PathStr) : Boolean;
```

Purpose

Add a file mask to a list of file masks.

Description

This function appends the specified file mask (FM) to the end of the list. It returns False if there is insufficient memory to add to the list.

FM can contain DOS wildcard characters ('*' or '?'), and it can also contain a path.

Example

See the example for FileMaskList.Match.

See Also

AppendFileMask

Syntax

```
function AppendFileMask(FM : PathStr; var FML : FileMaskList) : Boolean;
```

Purpose

Add a file mask to a list of file masks.

Description

This function adds a new file mask (FM) to the specified list of file masks (FML). It returns False if there is insufficient memory to create a new node in the list.

FM can contain DOS wildcard characters ('*' or '?'), and it can also contain a path.

Example

See the example for MatchFileMask.

See Also

FileMaskList.Append
InitFileMaskList

MatchFileMask

Syntax

```
function arOptionsAreOn(OptionFlags : Word) : Boolean;  
function Archive.arOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose

Return True if all specified options are on.

Description

This function returns True if the specified archive option(s) are currently selected.

Example

```
if arOptionsAreOn(arReadArcComments) then  
  arOptionsOff(arReadArcComments)  
else  
  arOptionsOn(arReadArcComments);
```

Toggle the state of the arReadArcComments option.

arOptionsOff / arOptionsOn / CreateOutputFile

Syntax

```
procedure arOptionsOff(OptionFlags : Word);  
procedure Archive.arOptionsOff(OptionFlags : Word);
```

Purpose

Deactivate multiple options.

Description

This procedure deactivates the specified archive option(s), excluding those designated as BadArchiveOptions. Note that, in APARCHIV, this routine modifies the typed constant DefArchiveOptions.

Example

See the example for arOptionsAreOn.

Syntax

```
procedure arOptionsOn(OptionFlags : Word);  
procedure Archive.arOptionsOn(OptionFlags : Word);
```

Purpose

Activate multiple options.

Description

This procedure activates the specified archive option(s), excluding those designated as BadArchiveOptions. Note that, in APARCHIV, this routine modifies the typed constant DefArchiveOptions.

Example

See the example for arOptionsAreOn.

Syntax

```
procedure CreateOutputFile(var F : File; FName : PathStr);  
procedure Archive.CreateOutputFile(var F : File; FName : PathStr);
```

Purpose

Create a file with the specified name.

Description

This routine creates a file of the specified name (FName) and returns its handle in F. If FName specifies a directory that doesn't exist, and the arCreateDirs option was turned on by calling arOptionsOn, CreateOutputFile tries to create the directory. If CreateOutputFile is successful, the file is open on return.

CreateOutputFile is intended primarily for internal use.

Examples

```
arOptionsOn(arCreateDirs);  
CreateOutputFile(F, 'C:\FILES\SUBDIR\MYPROG.PAS');  
if ArchiveStatus <> 0 then Halt;
```

This example creates a file named C:\FILES\SUBDIR\MYPROG.PAS. If C:\FILES\SUBDIR doesn't exist, CreateOutputFile tries to create it. Note that if C:\FILES doesn't exist, CreateOutputFile tries to create it first, then the SUBDIR subdirectory.

```
A.arOptionsOn(arCreateDirs);  
A.CreateOutputFile(F, 'C:\FILES\SUBDIR\MYPROG.PAS');  
if A.GetLastError <> 0 then Halt;
```

This is the OOP equivalent of the previous example.

See Also

arOptionsOn

GetLastError

Syntax

destructor Archive.Done; virtual;

Purpose

Destroy the archive object and close the input file.

Description

This destructor closes the input file previously opened by Archive.Init.

See Also

Archive.Init

Syntax

destructor FileMaskNode.Done; virtual;

Purpose

Deallocate memory used by the node.

Description

This destructor deallocates memory previously allocated by FileMaskNode.Init.

Example

See the example for FileMaskNode.Match.

See Also

FileMaskNode.Init

Syntax

destructor FileMaskList.Done; virtual;

Purpose

Deallocate memory used by all nodes in the list.

Description

This destructor deallocates memory used by all the nodes in the list.

Example

See the example for FileMaskList.Match.

See Also

DoneFileMaskList

FileMaskList.Init

Syntax

procedure DoneFileMaskList(var FML : FileMaskList);

Purpose

Deallocate memory used by a FileMaskList.

Description

This procedure deallocates memory used by all nodes in the specified FileMaskList.

Example

See the example for MatchFileMask.

See Also

FileMaskList.Done
InitFileMaskList

MatchFileMask

ExpandFileMaskList / GetFileName / GetLastError

Syntax

```
procedure ExpandFileMaskList(var FML, ExpandedFML : FileMaskList);  
procedure FileMaskList.ExpandFileMaskList;
```

Purpose

Expand all masks into complete file names.

Description

This procedure expands a list of file masks into a list of specific file names. All Async Professional LZH compressor routines call this routine prior to beginning the compress or delete process.

A file mask without an extension is assumed to be a file mask with a wildcard extension (e.g., 'TEXT' is assumed to be 'TEXT.*') since this is the way DOS operates on file masks. If you don't want this interpretation of a mask, then simply supply a trailing period (e.g., 'TEXT.' will match only the file named TEXT).

Note that the FileMaskList method ExpandFileMaskList expands the file mask internally, without creating a second FileMaskList. The non-OOP function ExpandFileMaskList, on the other hand, requires you to supply a second (initialized) file mask list--ExpandedFML--which it fills with the expanded file names. ExpandFileMaskList does not dispose of the file mask list; it is the caller's responsibility to do that.

Syntax

```
function Archive.GetFileName : PathStr;
```

Purpose

Get name of an archive.

Description

This method returns the name of the input file previously opened by Init.

Example

```
A.Init('FOO.ZIP');  
...  
WriteLn(A.GetFileName);
```

This displays 'FOO.ZIP'.

See Also

Init

Syntax

```
function Archive.GetLastError : Word;
```

Purpose

Get the code for the last error.

Description

This function returns the code for the last error that the Archive encountered while performing file I/O or attempting to allocate memory from the heap. GetLastError also clears the Archive's internal error variable.

Example

See the example for CreateOutputFile.

See Also

CreateOutputFile

Syntax

constructor Archive.Init(FName : PathStr);

Purpose

Initialize the archive and open the input file.

Description

This constructor opens the specified file (FName) and initializes the Archive's internal data fields. Errors are reported via the ArchiveStatus variable. The most likely error is ecFileNotFound (2).

Note that the mode in which the file is opened depends on the current setting of Turbo Pascal's FileMode variable. If you want to open the file in read-only mode, set FileMode to \$20 before calling Init.

See Also

UnLzh.Init

UnZip.Init

Syntax

constructor FileMaskNode.Init(FM : PathStr);

Purpose

Initialize the node.

Description

This constructor initializes the node. It splits the specified file specification (FM) into its component parts and stores them in the FileMaskNode.

Example

See the example for FileMaskNode.Match.

Syntax

constructor FileMaskList.Init;

Purpose

Initialize a file mask list.

Description

This constructor initializes an empty file mask list.

Example

See the example for FileMaskList.Match.

Syntax

procedure InitFileMaskList(var FML : FileMaskList);

Purpose

Initialize a file mask list.

Description

This procedure initializes the FileMaskList FML. It should be called prior to any calls to AppendFileMask.

Example

See the example for MatchFileMask.

See Also

AppendFileMask
DoneFileMaskList

FileMaskList.Init
MatchFileMask

Match / Match

Syntax

```
function FileMaskNode.Match(Dir : DirStr; Name : NameStr;  
                           Ext : ExtStr) : Boolean;
```

Purpose

Return True if the specified file name matches the file mask stored in this node.

Description

This function returns True if the file mask represented by Dir, Name, and Ext matches the file mask stored in this node. Note that it is assumed that Ext does not contain a '!'.

Example

```
var  
  FMN : FileMaskNode;  
begin  
  FMN.Init('*.*PAS');  
  WriteLn( FMN.Match('', '*', 'PAS') );  
  WriteLn( FMN.Match('', 'MY*', 'PAS') );  
  WriteLn( FMN.Match('', 'MYPROG', 'PAS') );  
  FMN.Done;  
end.
```

This program writes 'True' to the screen three times.

See Also

FileMaskList.Match

Syntax

```
function FileMaskList.Match(FName : PathStr) : Boolean;
```

Purpose

Return True if the specified file name is in list of file masks.

Description

This function, intended primarily for internal use, returns True if the specified filename or file mask (FName) matches one of the file masks stored in the list. FName can contain DOS wildcard characters ('*' or '?'), and it can also specify a pathname. Match returns True if the file mask list is empty.

Example

```
var  
  FML : FileMaskList;  
begin  
  FML.Init;  
  if not FML.Append('*.*PAS') then Halt;  
  WriteLn( FML.Match('MY*.*PAS') );  
  WriteLn( FML.Match('MYPROG.*PAS') );  
  WriteLn( FML.Match('FILES\MYPROG.*PAS') );  
  FML.Done;  
end.
```

This program writes 'True' to the screen twice, and then 'False'.

See Also

MatchFileMask

Syntax

```
function MatchFileMask(FM : PathStr; var FML : FileMaskList) : Boolean;
```

Purpose

Return True if the specified file name is in the list of file masks.

Description

This function, intended primarily for internal use, returns True if the specified filename or file mask (FM) matches one of the file masks stored in FML. FM can contain DOS wildcard characters ('*' or '?'), and it can also specify a pathname. MatchFileMask returns True if the file mask list is empty.

Example

```
var
  FML : FileMaskList;
begin
  InitFileMaskList(FML);
  if not AppendFileMask('*.PAS', FML) then Halt;
  WriteLn( MatchFileMask('MY*.PAS', FML) );
  WriteLn( MatchFileMask('MYPROG.PAS', FML) );
  WriteLn( MatchFileMask('FILES\MYPROG.PAS', FML) );
  DoneFileMaskList(FML);
end.
```

This program writes 'True' to the screen twice, and then 'False'.

See Also

AppendFileMask
InitFileMaskList

FileMaskList.Match

Syntax

```
procedure SetOutputPath(Path : PathStr);
procedure Archive.SetOutputPath(Path : PathStr);
```

Purpose

Set a path for output files.

Description

This procedure lets you specify the drive and directory where files are to be written when they are dearchived. In APARCHIV, this path is stored in the typed constant arOutPath. In OOARCHIV, it is stored in an internal data field of the object.

Example

```
SetOutputPath('C:');
```

Request that dearchived files be written to the current directory of drive C.

Syntax

```
procedure SortFileMaskList(var FML : FileMaskList);
procedure FileMaskList.SortFileMaskList; virtual;
```

Purpose

Sorts the file masks in a file mask list in ascending order.

Description

This routine sorts the file masks, or expanded file names, in an existing FileMaskList. The sort order is ascending. All Async Professional LZH compressor routines call this procedure after expanding the file masks so that the files can be processed in alphabetical order.

The existing file mask list is sorted in place (no separate output FileMaskList is required).

C. LZH Archives: APLZH/OOLZH

These two units provide facilities for working with LZH archives. Using the routines/methods in them, you can archive files to an LZH archive, list the files stored in an LZH archive, and extract files from an LZH archive.

The LZH object hierarchy is:

```
Archive
├── UnLzh
│   └── Lzh
```

The Lzh object, which compresses files to an LZH archive, is derived from the UnLzh object, which decompresses files from an LZH archive. In this way, the Lzh object can share much of the "boilerplate" code in the UnLzh object responsible for reading and writing file headers, calculating CRCs, and so on. The only downside to this hierarchy is that it isn't possible to have a "compress only" object—your Lzh objects will always include a few virtual methods that apply only to decompression.

APLZH/OOLZH consume approximately 16K of data segment space for initialized and uninitialized tables. (These tables are stored in the data segment for faster access during compression and decompression.) This is quite a bit of data segment space and may push you over the 64K limit if you are already making substantial use of global variables. The only solution to this problem is to move some of your global variables to the heap. APLZH/OOLZH's use of the data segment is fundamental to their design and can't easily be changed.

APLZH/OOLZH also require approximately 42K of heap space for decompression buffers and, when compressing, an additional 153K of heap space for temporary tables. This heap space is allocated at the start of all compression and extraction routines (e.g., CompressFileMaskList and ExtractFileMaskList) and is disposed before the routine returns.

And finally, each node of a file mask list also consumes heap space. Such heap usage may become significant if your file mask list expands into several hundred specific file names. Each node requires 24 bytes of heap space plus the amount required to store the directory name (zero bytes if no directory information is specified).

APLZH and OOLZH work with LZH files created with LHARC version 1.x or LHA version 2.x. Note that APLZH and OOLZH *do* work with self-extracting files created with LHARC's S option, as well as with regular LZH files.

The remainder of this section discusses how to view an LZH archive directory, how to extract files, how to compress files, the available user hooks, and the structure of an LZH archive.

Viewing the Directory of an LZH Archive

The following sample program, LZHLIST, illustrates the basic steps involved in viewing the directory of an LZH file. LZHLIST takes the name of the LZH file to view as a command line parameter. If the file is opened successfully and is indeed an LZH archive, it displays the names and the (uncompressed) sizes of all the files in the archive.

```
program LzhList; {LZHLIST.PAS}
uses ApMisc, ApArchiv, ApLzh;
var
  FML : FileMaskList;
  LFL : LzhFileList;
  LNP : LzhNodePtr;
begin
```

```

{create an empty file mask list}
InitFileMaskList(FML);

{initialize LZH file list}
InitLzhFileList(LFL);

{open the LZH file}
InitLzhFile( ParamStr(1) );
if ArchiveStatus <> 0 then begin
  Writeln('Error: ', StatusStr(ArchiveStatus));
  Halt;
end;

{enable user hooks}
SetShowNameProc(DefShowNameProc);

{construct list of files in archive}
BuildLzhFileList(LFL, FML);
if ArchiveStatus <> 0 then begin
  Writeln('Error: ', StatusStr(ArchiveStatus));
  Halt;
end;

{display names and sizes of all files in archive}
LNP := LFL.Head;
while LNP <> nil do begin
  with LNP^.LH do
    Writeln(OrigSize:7, ' ', FName);
  LNP := LNP^.Next;
end;

{close the LZH file}
DoneLzhFile;

{dispose of data structures}
DoneLzhFileList(LFL);
DoneFileMaskList(FML);
end.

```

The first step here is to create an empty file mask list, to be passed later to BuildLzhFileList, by calling InitFileMaskList. In a more sophisticated listing program, such as LZHV, you would probably want to give the user some means of specifying one or more file masks that would be added to the list. In this case, though, the list is empty, making it equivalent to a list with a single mask of *.*.

Step two is to initialize a variable of type LzhFileList (LFL). The actual list will be created by BuildLzhFileList when it scans the directory of the LZH file.

Step three is to open the LZH file by calling InitLzhFile. This routine tries to open the specified file and determine if it is an LZH archive. If there are no errors in opening or reading from the file, the only possible error is ecNotAnLzhFile.

Step four is an optional one: SetShowNameProc is called to take advantage of a hook that allows you to display an appropriate message just before BuildLzhFileList starts scanning the LZH's directory. The supplied ShowName procedure, DefShowNameProc, simply displays 'Searching' followed by the name of the LZH file.

Step five is to use BuildLzhFileList to construct a list of all files in the archive that match at least one of the masks in the file mask list. The resulting list (LFL) is singly-linked, and each node in the list is of type LzhNode (the following declaration is from APLZH):

```

LzhNode =
record
    Next      : LzhNodePtr;
    LH        : LzhHeader;
    FileOfs   : LongInt;
    Tagged    : Boolean;
end;

```

For the moment, the only fields of interest here are Next, which points to the next node in the list (or is nil, if there are no more nodes), and LH, the LZH's directory entry for a particular file.

Step six is to go through all the nodes in the list and write out the size (uncompressed) and name of each file. This information is stored in the LH field of the node, a variable of type LzhHeader:

```

LzhHeader =
record
    HeadSize      : Byte;           {size of header}
    HeadChk       : Byte;           {checksum for header}
    HeadID        : HeadIDType;     {compression type tag}
    NewSize       : LongInt;        {compressed size}
    OrigSize      : LongInt;        {original size}
    Time         : Word;            {packed time}
    Date         : Word;            {packed date}
    Attr         : Byte;            {file attributes}
    Level        : Byte;            {=0 LZH method, =1 LHA method}
    FName        : PathStr;         {filename (variable length)}
    CRC          : Word;            {16-bit CRC (immediately follows FName)}
    OSID         : Char;            {=M for DOS - LHA method}
    PathHdrSize  : Word;            {path extended header size}
    PathHdrID    : Byte;            {=2 "Path" extended header flag}
    ExtFPath     : PathStr;         {pathname (variable length)}
    AttrHdrSize  : Word;            {size of the attribute header}
    AttrHdrID    : Byte;            {attribute header ID}
    ExtAttr      : Word;            {extended attribute}
    FNameHdrSize : Word;            {filename extended header size}
    FNameHdrID   : Byte;            {=1 "FileName" extended header}
    ExtFName     : PathStr;         {filename (variable length)}
    CRCHdrSize   : Word;            {=5 extended CRC Header}
    CRCHdrID     : Byte;            {=0 "CRC" extended header flag}
    ExtCRC       : Word;            {extended Header CRC value}
    NextHdrSize  : Word;            {=0 No more extended headers}
end;

```

In this simple example, only the OrigSize and FName fields are displayed, but in a more sophisticated listing program, such as LZHV, almost all of them would be displayed. See LZHV.PAS for an example of how to interpret and format the other fields in the record.

The final two steps are to close the LZH file and to dispose of the data structures (LFL and FML). Although these steps are technically unnecessary in this particular program, they would be necessary in a program that did something other than halt after it was finished with the LZH file.

The object-oriented version of the program, LZHLISTO, is quite similar:

```

program LzhListO; {LZHLISTO.PAS}
uses
    ApMisc, OoArchiv, OoLzh;
var
    UL : UnLzh;
    FML : FileMaskList;

```

```

LFL : LzhFileList;
LNP : LzhNodePtr;
begin
  {create an empty file mask list}
  FML.Init;

  {initialize LZH file list}
  LFL.Init;

  {open the LZH file}
  if not UL.Init( ParamStr(1) ) then begin
    Writeln('Error: ', StatusStr(ArchiveStatus));
    Halt;
  end;

  {enable user hooks}
  UL.SetShowNameProc(DefShowNameProc);

  {construct list of files in archive}
  UL.BuildLzhFileList(LFL, FML);
  if UL.GetLastError <> 0 then begin
    Writeln('Error: ', StatusStr(UL.GetLastError));
    Halt;
  end;

  {display names and sizes of all files in archive}
  LNP := LFL.lfHead;
  while LNP <> nil do begin
    with LNP^.lnLH do
      Writeln(OrigSize:7, ' ', FName);
    LNP := LNP^.lnNext;
  end;

  {close the LZH file}
  UL.Done;

  {dispose of data structures}
  LFL.Done;
  FML.Done;
end.

```

The only notable points of difference between the second program and the first are 1) you use the `GetLastError` method rather than `ArchiveStatus` to check for errors, and 2) the names of data fields in `LzhNodes` and `LzhFileLists` are preceded by prefixes, 'lf-' for `LzhFileList` and 'ln-' for `LzhNode`. The declaration for the `LzhHeader` type in `OOLZH` is identical to that in `APLZH`.

Extracting Files from an LZH Archive

The process of extracting files from an LZH archive is largely identical to that used for viewing its directory, as the following example program shows. The principal difference is that you rarely need to work with LZH file lists. More often, you can simply extract files based on a file mask list, as is done here:

```

program LzhExt; {LZHEXT.PAS}
uses
  ApMisc, ApArchiv, ApLzh;
var
  FML : FileMaskList;
begin
  {create an empty file mask list}
  InitFileMaskList(FML);

```

```

{open the LZH file}
InitLzhFile( ParamStr(1) );
if ArchiveStatus <> 0 then begin
  Writeln('Error: ', StatusStr(ArchiveStatus));
  Halt;
end;

{enable user hooks}
SetShowNameProc(DefShowNameProc);
SetShowMethodProc(DefShowMethodProc);
SetExtractSuccessFunc(DefExtractSuccessFunc);
SetShowProgressFunc(DefShowProgressFunc);

{extract all files in archive}
ExtractFileMaskList(FML);
if ArchiveStatus <> 0 then begin
  Writeln('Error: ', StatusStr(ArchiveStatus));
  Halt;
end;

{close the LZH file}
DoneLzhFile;

{dispose of data structures}
DoneFileMaskList(FML);
end.

```

As in LZHLIST, step one is to create an empty file mask list. Step two is to open the LZH file, whose name is specified by the user as a command line parameter.

Step three is to enable a series of user-written routines that are called at various points in the extraction process to display status information, display error messages, abort the extraction operation, etc. APLZH provides seven such hooks, four of which are used here. For a discussion of these hooks, see "APLZH/OOLZH User Hooks" later in this section.

Step four is to call ExtractFileMaskList to extract the files from the library. This routine looks at each entry in the LZH's directory to see if it matches one of the masks in the specified file mask list. If it does, the file is extracted.

As in LZHLIST, the final two steps are to close the LZH file and to dispose of the data structures. Here again, these steps are technically unnecessary in this particular program.

LZHEXTO, the object-oriented version of LZHEXT, is quite similar to the procedural version:

```

program LzhExtO {LZHEXTO.PAS}
uses
  ApMisc, OoArchiv, OoLzh;
var
  UL : UnLzh;
  FML : FileMaskList;
begin
  {create an empty file mask list}
  FML.Init;

  {open the LZH file}
  if not UL.Init( ParamStr(1) ) then begin
    Writeln('Error: ', StatusStr(ArchiveStatus));
    Halt;
  end;

  {enable user hooks}

```



```

UL.SetShowNameProc(DefShowNameProc);
UL.SetShowMethodProc(DefShowMethodProc);
UL.SetExtractSuccessFunc(DefExtractSuccessFunc);
UL.SetShowProgressFunc(DefShowProgressFunc);

{extract all files in archive}
UL.ExtractFileMaskList(FML);
if UL.GetLastError <> 0 then begin
    Writeln('Error: ', StatusStr(UL.GetLastError));
    Halt;
end;

{close the LZH file}
UL.Done;

{dispose of data structures}
FML.Done;
end.

```

The only point worth noting here is that the program could be further simplified by getting rid of the FileMaskList and calling

```
UL.Extract('*.');
```

rather than UL.ExtractFileMaskList.

Selecting Files to be Extracted

When extracting files from an LZH archive, there are three ways that you can select the files:

1. File masks. This approach is the one used in the example program in the previous section. A list of file masks is created and then passed to ExtractFileMaskList.
2. Building an LzhFileList and deleting nodes. This approach involves using BuildLzhFileList to construct a list of all the files stored in the archive (see the example program LZHLIST, above). Once this list is created, you use DeleteLzhFileListNode (APLZH) or LzhFileList.Delete (OOLZH) to remove all files that you don't want to extract. You then call ExtractLzhFileList to extract.
3. Building an LZH file list and tagging/untagging nodes. This approach is similar to the second, in that you start by calling BuildLzhFileList to construct a list of files in the archive. But instead of removing a given node from the list, you can request that it not be extracted simply by setting its tag field to False (the default value is True), as follows:

APLZH	OOLZH
var LNP : LzhNodePtr;	var LNP : LzhNodePtr;
...	...
LNP^.Tagged := False;	LNP^.lnTagged := False;

Once you've untagged all the files that you don't want extracted, you can call ExtractFileList to extract the tagged files. This approach is particularly well-suited to applications that display a list of all the files in an archive and allow the user to interactively select the files to be extracted.

Updating an LZH Archive

Three types of compression routines are available in APLZH and OOLZH:

Compress - new files are added and specified old files are updated in an existing or new archive file.

Delete - specified files are removed from an existing archive file.

Freshen - every file within an archive is compared to its unarchived version on disk and is updated in the archive if the unarchived version has a later date/time stamp.

When updating an archive in any of these ways, a new archive file is created and files are copied from the old archive, or added new, until all files are processed. The original archive is renamed to an extension of ')1(' and the new archive is given an extension of ')2('. Files are added to the new archive in alphabetical order.

When all files are processed, the new ')2(' output file is renamed to the original archive file name and the old ')1(' archive file is deleted. Should the update operation fail at any point, the ')1(' file is renamed back to its original name and the new ')2(' file is deleted.

Obviously, the update process fails if there is not enough disk space to hold both the old archive file and the new archive file. Since there's no foolproof way to predict how big the resulting archive will be, your program must be prepared to deal with `ecDiskFull` errors after calling an update routine. Although the original archive file is restored, your application has to decide how to proceed when it receives this error.

Just as you build a list of file masks when extracting files, you also build a list of file masks when compressing or deleting files. Here's a bare-bones example of a non-OOP program that shows this:

```
program ExLzh;
uses
  Dos, ApMisc, ApArchiv, ApLzh;
var
  FML : FileMaskList;

begin
  {Initialize the file mask list and add masks}
  InitFileMaskList(FML);
  if not AppendFileMask('EXLZH.PAS', FML) or
    not AppendFileMask('EXLZH.EXE', FML) then begin
    WriteLn('Insufficient memory');
    Halt;
  end;

  {Create a new LZH file}
  CreateLzhFile('EXLZH.LZH');
  if ArchiveStatus <> ecOk then begin
    WriteLn('Failed to create archive, error: ', ArchiveStatus);
    Halt;
  end;

  {Set options}
  SetShowMethodProc(DefShowMethodProc);
  SetShowProgressFunc(DefShowProgressFunc);
  SetShowNameProc(DefShowNameProc);

  {Set compressing-only options}
  SetOkToCompressFunc(DefOkToCompressFunc);
  SetCompressSuccessFunc(DefCompressSuccessFunc);

  {Add the files the archive}
  CompressFileMaskList(FML);

  {Report errors}
  if ArchiveStatus <> ecOk then
```

```

        WriteLn('Failed due to error ', ArchiveStatus mod 10000);

    {Clean up}
    DoneFileMaskList(FML);
    DoneLzhFile;
end.

```

And here's the OOP version of the same program:

```

program ExLzh0;
uses
    Dos, ApMisc, OoArchiv, OoLzh;
var
    FML : FileMaskList;
    L : Lzh;
    Result : Word;
begin
    {Instantiate the file mask list and add masks}
    FML.Init;
    if not FML.Append('EXLZHO.PAS') or
       not FML.Append('EXLZHO.EXE') then begin
        WriteLn('Insufficient memory');
        Halt(1);
    end;

    {Create LZH file}
    L.Create('EXLZHO.LZH');
    if ArchiveStatus <> ecOk then begin
        WriteLn('Failed to create archive, error: ', ArchiveStatus);
        Halt;
    end;

    {Set standard options}
    L.SetShowMethodProc(DefShowMethodProc);
    L.SetShowProgressFunc(DefShowProgressFunc);
    L.SetShowNameProc(DefShowNameProc);

    {Set compressing-only options}
    L.SetOkToCompressFunc(DefOkToCompressFunc);
    L.SetCompressSuccessFunc(DefCompressSuccessFunc);

    {Add the files to the archive}
    L.CompressFileMaskList(FML);

    {Report errors}
    Result := L.GetLastError;
    if Result <> ecOk then
        WriteLn('Failed due to error ', Result);

    {Clean up}
    FML.Done;
    L.Done;
end.

```

The first step in both of these programs is to build the list of files to be added to the archive. In each case, just the .PAS and .EXE files of the example program are added (i.e., EXLZH.PAS and EXLZH.EXE). If the file masks are added successfully, a new archive file is created. Note that the program fails if the archive file already exists. In a more sophisticated program (like the supplied LZH and LZHO) you would probably want to handle this situation more gracefully.

After creating the archive file, the various user hooks to the Async Professional built-in routines are set. The first group of user hooks is also used when extracting files. The last two user hooks are specific to compressing files and are described in more detail below. For now, it's enough to know that the `OkToCompress` function is called to verify that it's OK to compress each matching file. The `CompressSuccess` function is called after the file is compressed so you can update your status display with the compression results.

Next, the example program calls `CompressFileMaskList` to compress all files that meet the masks in FML. In this case, FML contains exact file names rather than wildcard masks, but it works the same in either case. Files that match the masks are compressed into the archive. The files are compressed using the LHA freezing method. To specify a different compression method, use `SetCompressionMode` (see the description later in this section).

If any errors occur during the compression process, `ArchiveStatus` (or `L.GetLastError`) will contain a code describing the cause of the error. Finally, the file mask list is disposed and the archive file is closed by a call to `DoneLzhFile` (or `L.Done`).

APLZH/OOLZH User Hooks

APLZH and OOLZH provide seven hooks that allow you to designate user-written functions that are called at specific points during the processing of an LZH file. Three of these user hooks (`ShowName`, `ShowMethod`, and `ShowProgress`) are potentially useful in all situations. Two of them (`OkToWrite` and `ExtractSuccess`) are useful only in programs that extract files. And the final two (`OkToCompress` and `CompressSuccess`) are useful only in programs that compress files.

A `ShowName` procedure is called just before an LZH's directory is scanned in search of the names of the files in it. This gives you an opportunity to display a "Searching..." message after the file has been opened and determined to be a valid LZH archive. When compressing, the default routine `DefShowNameProc` displays 'Updating' and the LZH file name. Otherwise, it displays 'Searching' and the LZH file name. See `SetShowNameProc` for details.

An `OkToWrite` function is called before a file is extracted to give you an opportunity to prevent the operation (e.g., because the file already exists and either the existing file is newer than the archived one or the user doesn't want it overwritten). An `OkToWrite` function can also change the name of the file to be written (e.g., in cases where the file already exists, and you want to extract it under another name). The demonstration programs `LZHX` and `LZHXO` contain sophisticated examples of `OkToWrite` functions. See `SetOkToWriteFunc` for details.

A `ShowMethod` procedure is called just before a file is compressed or extracted, giving you an opportunity to display the name of the file and the compression method used on it. See `SetShowMethodProc` for details. Despite the name, the real purpose of this routine during compression is to display the current file name. If you want to display the compression method, you'll need to do it in the `CompressSuccess` function because the compression method isn't known until the compression is finished. The default routine, `DefShowMethodProc`, displays the file name on the current line and the string " Freezing : " on the line below it. When deleting files, `DefShowMethodProc` displays " Deleting : " and the filename on the same line.

A `ShowProgress` function is called regularly during the compression and extraction processes. This hook gives you an opportunity to 1) indicate to the user that progress is being made and 2) abort the operation if desired. The default routine, `DefShowProgressFunc`, displays dots ('.') during extraction and o's ('o') during compression. See `SetShowProgressFunc` for details.

An `ExtractSuccess` function is called after attempting to extract a file. This hook gives you an opportunity to display an error message if an error occurred, and to abort the extraction operation if

desired. See `SetExtractSuccessFunc` for details. Note that this function is not called when compressing files.

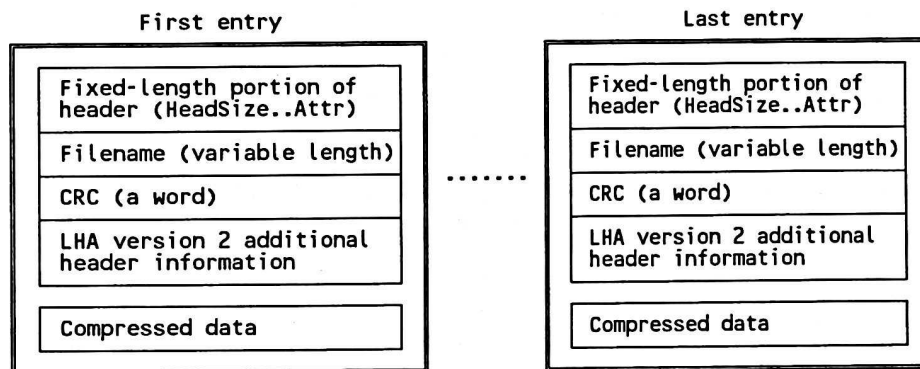
The `OkToCompress` hook is a boolean function that is called whenever an update routine finds a file that might need to be added to an archive or updated. Your `OkToCompress` function must determine if the specified file should be added to the archive, or if an existing file should be replaced in the archive. See `SetOkToCompressFunc` for details.

The `CompressSuccess` function is very similar in form and function to the `ExtractSuccess` function described above. This function is called by all update routines (compress or freshen) after each attempt to compress a file. Generally, the role of this function is simply to announce that the compress process is over and, perhaps, display a compression ratio or other information. See `SetCompressSuccessFunc` for details.

Structure of an LZH File

In general, you don't need to know anything about the structure of an LZH file in order to use the routines in `APLZH` and `OOLZH`. But you may find the information in this section useful when you want to do something with LZH files that `Async Professional` doesn't support directly.

Each entry in an LZH archive is stored in two parts: a header describing the file, followed by the data for the compressed file. An LZH archive as a whole is simply a series of such entries:



The actual structure of the header is in the declaration for the `LzhHeader` type.

An ordinary LZH file begins with the first entry and ends with the last. A self-extracting LZH file has code and data before the first entry, however, making it necessary to scan the file in search of an LZH header ID. (See the source code for `InitLzhFile` in `APLZH.PAS`.)

Shared Declarations

Constants

```

cmcStored = '0'; {stored (no compression)}
cmcFrozen1 = '1'; {shrunk by LHARC 1.x}
cmcFrozen2 = '5'; {shrunk by LHA 2.x}
  
```

The possible compression method codes. In an `LzhHeader`, the compression method code is at `HeadID[4]`.

```

NextDisplayInc : Word;
  
```

Controls how frequently the `ShowProgress` function is called. `NextDisplayInc` is the number of bytes that should be read from the source file and compressed before the `ShowProgress` function is called again. The default is 4096 bytes.

Types

```

LzhHeader =
record
    HeadSize      : Byte;           {size of header}
    HeadChk       : Byte;           {checksum for header}
    HeadID        : HeadIDType;     {compression type tag}
    NewSize       : LongInt;        {compressed size}
    OrigSize      : LongInt;        {original size}
    Time         : Word;            {packed time}
    Date         : Word;            {packed date}
    Attr         : Byte;            {file attributes}
    Level        : Byte;            {=0 LZH method, =1 LHA method}
    FName        : PathStr;         {filename (variable length)}
    CRC          : Word;            {16-bit CRC (immediately follows FName)}
    OSID         : Char;            {=M for DOS - LHA method}
    PathHdrSize  : Word;            {path extended header size}
    PathHdrID    : Byte;            {=2 "Path" extended header flag}
    ExtFPath     : PathStr;         {pathname (variable length)}
    AttrHdrSize  : Word;            {size of the attribute header}
    AttrHdrID    : Byte;            {attribute header ID}
    ExtAttr      : Word;            {extended attribute}
    FNameHdrSize : Word;            {filename extended header size}
    FNameHdrID   : Byte;            {=1 "FileName" extended header}
    ExtFName     : PathStr;         {filename (variable length)}
    CRCHdrSize   : Word;            {=5 extended CRC Header}
    CRCHdrID     : Byte;            {=0 "CRC" extended header flag}
    ExtCRC       : Word;            {extended Header CRC value}
    NextHdrSize  : Word;            {=0 No more extended headers}
end;

```

Data structure similar to the header that precedes a compressed file stored in an LZH archive. In an actual file header, some of the fields are variable length. The header is then followed by the compressed file itself.

```
CompressionMode = (cmBestMethod, cmStored, cmFrozen1, cmFrozen2);
```

Enumerated type used to select the compression mode.

APLZH Declarations

Types

```
CompressSuccessFunc = function(var LH : LzhHeader; ErrCode : Word) : Boolean;
```

A function of this type is called after each file is compressed, giving you a chance to update your status display. LH is the header for the file being extracted. The function should return True if it wants the compression operation to continue.

```
ExtractSuccessFunc = function(var LH : LzhHeader; FName : PathStr;
                               ErrCode : Word) : Boolean;
```

A function called after a file is extracted from an archive. ErrCode contains 0 if the extraction was successful, else an error code. LH is the header for the file being extracted, and FName is the name of the output file that was created. The function should return True if it wants the extraction operation to continue.

```

LzhFileList =
record
    Head, Tail : LzhNodePtr;
    Count : Word;
end;

```

A singly-linked list of files in an LZH file, generally created by calling BuildLzhFileList. Head points to the first node in the list, Tail to the last. Count is the number of nodes in the list.

```

LzhNodePtr = ^LzhNode;
LzhNode = record
    Next      : LzhNodePtr;
    LH        : LzhHeader;
    FileOfs   : LongInt;
    Tagged    : Boolean;
end;

```

A single node in an LzhFileList. FileOfs is the offset within the LZH file for the compressed file described by the LzhHeader. Next points to the next node in the list. Tagged can be set to False to prevent the file from being extracted.

```

OkToCompressFunc = function(NewFile : PathStr; LH : LzhHeader) : Boolean;

```

A function of this type is called before each file is compressed, giving you a chance to skip the compression.

```

OkToWriteFunc = function(var LH : LzhHeader; var FName : PathStr) : Boolean;

```

A function of this type is called to determine whether or not the specified file (FName) should be extracted from the LZH archive. This function can also change the name of the file to be written. Note that the call is made after any file mask matching has been done.

```

ShowMethodProc = procedure(Method : Char; FName : PathStr);

```

A procedure of this type is called just before the specified file (FName) is compressed or extracted, giving you an opportunity to display its name and (during extraction) the compression method used (cmcStored, cmcFrozen1, or cmcFrozen2).

```

ShowNameProc = procedure(FName : PathStr);

```

A procedure of this type is called when an LzhFileList is being built, giving you an opportunity to display the name of the LZH file.

```

ShowProgressFunc = function(BytesWritten, TotalBytes : LongInt) : Boolean;

```

A function of this type is called periodically during the compression or extraction of a file, giving you an opportunity to indicate to the user that progress is being made. It should return False if it wants to abort the extract operation.

Variables

```

Crc : Word;

```

Running CRC counter. This variable can be used by an ExtractSuccess function in the event of an ecBadFileCRC error to determine what the computed CRC value was. (LH.CRC would contain the correct CRC value.)

OOLZH Declarations

Types

```

CompressSuccessFunc = function(LP : LzhPtr; LH : LzhHeader) : Boolean;

```

A function of this type is called after each file is compressed, giving you a chance to update your status display. LH is the header for the file being extracted. The function should return True if it wants the compression operation to continue.

```

ExtractSuccessFunc = function(LNP : LzhNodePtr; FName : PathStr;
                               UP : UnLzhPtr) : Boolean;

```

A function called after a file is extracted from an archive. LNP points to the LzhNode corresponding to the file being extracted, and FName is the name of the output file that was being created. The success of the operation can be determined by calling UP^.GetLastError. The function should return True if it wants the extraction operation to continue.

```

LzhFileListPtr = ^LzhFileList;
LzhFileList =
  object
    lfHead : LzhNodePtr;
    lfTail : LzhNodePtr;
    lfCount : Word;
    ...
  end;

```

A singly-linked list of files in an LZH file, created by calling `UnLzh.BuildLzhFileList`. `lfHead` points to the first node in the list, `lfTail` to the last. `lfCount` is the number of nodes in the list.

```

LzhNodePtr = ^LzhNode;
LzhNode =
  object
    lnNext : LzhNodePtr;
    lnLH : LzhHeader;
    lnFileOfs : LongInt;
    lnTagged : Boolean;
    ...
  end;

```

A single node in an `LzhFileList`. `lnFileOfs` is the offset within the LZH file for the compressed file described by the `LzhHeader`. `lnNext` points to the next node in the list. `lnTagged` can be set to `False` to prevent the file from being extracted.

```

OkToCompressFunc = function(LP : LzhPtr; NewFile : PathStr;
  LH : LzhHeader) : Boolean;

```

A function of this type is called before each file is compressed, giving you a chance to skip the compression.

```

OkToWriteFunc = function(LNP : LzhNodePtr; var FName : PathStr;
  UP : UnLzhPtr) : Boolean;

```

A function of this type is called to determine whether or not the specified file (`FName`) should be extracted from the LZH archive. This function can also change the name of the file to be written. Note that the call is made after any file mask matching has been done.

```

ShowMethodProc = procedure(LNP : LzhNodePtr; FName : PathStr; UP : UnLzhPtr);

```

A procedure of this type is called just before the specified file (`FName`) is compressed or extracted, giving you an opportunity to display its name and (during extraction) the compression method used (`cmcStored`, `cmcFrozen1`, or `cmcFrozen2`).

```

ShowNameProc = procedure(UP : UnLzhPtr);

```

A procedure of this type is called when an `LzhFileList` is being built, giving you an opportunity to display the name of the LZH file.

```

ShowProgressFunc = function(UP : UnLzhPtr;
  BytesWritten, TotalBytes : LongInt) : Boolean;

```

A function of this type is called periodically during the compression or extraction of a file, giving you an opportunity to indicate to the user that progress is being made. It should return `False` if it wants to abort the extract operation.

```

LzhPtr = ^Lzh;
Lzh =
  object(UnLzh)
    ...
  end;

```

An object used for compressing files into an LZH archive.


```

UnLzhPtr = ^UnLzh;
UnLzh =
    object(Archive)
        ulCrc : Word; {running CRC counter}
        ...
    end;

```

An object used for extracting information and/or compressed files from an LZH archive. ulCRC is a running CRC counter, used when extracting a file. This variable can be used by an ExtractSuccess function in the event of an ecBadFileCRC error to determine what the computed CRC value was.

Procedures and Functions

```

function Lzh.CompressSuccess(LH : LzhHeader) : Boolean; virtual;
    {-Called after file assoc. with LNP has been unarced}
function UnLzh.ExtractSuccess(LNP : LzhNodePtr;
                               FName : PathStr) : Boolean; virtual;
    {-Called after file assoc. with LNP has been unarced}
function Lzh.OkToCompress(NewFile : PathStr;
                           LH : LzhHeader) : Boolean; virtual;
    {-Returns True if OK to compress file NName}
function UnLzh.OkToWrite(LNP : LzhNodePtr;
                          var FName : PathStr) : Boolean; virtual;
    {-Returns True if OK to write file associated with LNP}
procedure UnLzh.ShowMethod(LNP : LzhNodePtr; FName : PathStr); virtual;
    {-Called to display name of file being unarced and comp. method}
procedure UnLzh.ShowName; virtual;
    {-Called to display name of LZH file}
function UnLzh.ShowProgress(BytesWritten, TotalBytes : LongInt)
                           : Boolean; virtual;
    {-Called when flushing output buffer to disk}

```

Virtual member functions for the OOLZH user hooks. You can override these virtual functions instead of using the SetXxx functions.

Append / BuildLzhFileList

Syntax

```
function LzhFileList.Append(var LH : LzhHeader; FO : LongInt) : Boolean;
```

Purpose

Add a node to the list.

Description

This method adds a new file to the end of the list, returning True if successful. LH is the LZH header for the file, and FO is its offset within the LZH file.

Append is intended primarily for internal use by UnLzh.BuildLzhFileList.

See Also

UnLzh.BuildLzhFileList

LzhNode.Init

Syntax

```
procedure BuildLzhFileList(var LFL : LzhFileList; var FML : FileMaskList);  
procedure UnLzh.BuildLzhFileList(var LFL : LzhFileList; var FML : FileMaskList);
```

Purpose

Build a list of files to be unarchived.

Description

This routine builds an LzhFileList containing all the files in the current LZH file that match one or more of the file masks in the specified FileMaskList. If no matching files are found, an error code of epWarning+ecNoMatchingFiles is generated.

Examples

See the examples for InitLzhFileList and LzhFileList.Init.

See Also

DoneLzhFileList

Archive.GetLastError

LzhFileList.Init

InitLzhFileList

Syntax

```
procedure CompressLzh(Mask : PathStr);  
procedure Lzh.Compress(Mask : PathStr);
```

Purpose

Compress all files that match a file mask.

Description

This is a simplified version of CompressFileMaskList. You may find this format more convenient when all of the files to be compressed can be described by a single mask.

See CompressFileMaskList for more information.

Syntax

```
procedure CompressFileMaskListLzh(var FML : FileMaskList);  
procedure Lzh.CompressFileMaskList(var FML : FileMaskList);
```

Purpose

Compress all files that match the file mask list.

Description

This routine compresses all files that match one or more of the file masks in the specified FileMaskList.

LZH compression includes three formats of compression: the LHA freezing method, the LHARC freezing method, and storing, where the file is simply stored as-is in the archive.

By default, CompressFileMaskList attempts to compress a file using the freezing method used by LHA 2.x. If the compressed file size is larger than the original file size, then the compressed image is discarded and the file is simply copied into the archive. If you want to use a different compression method, call SetCompressionMode to specify it before you call CompressFileMaskList.

Examples

See CreateLzhFile for a small example. See the EXLZH and EXLZHO programs (at the beginning of §11.C) for simple but complete examples of using CompressFileMaskList.

See Also

Compress

SetCompressionMode

Create

Syntax

constructor Lzh.Create(FName : PathStr);

Purpose

Create a new LZH archive file.

Description

This procedure creates a new LZH archive file. It creates and initializes a new LZH archive using the standard LZH format used by LHARC and LHA. Use Create in lieu of Init when instantiating an Lzh object associated with a new archive file.

An extension of LZH is assumed if none is specified.

This procedure fails with ecCannotCreate if an archive with the specified name already exists. If any DOS I/O errors are encountered, the procedure fails and ArchiveStatus contains the DOS error code.

If you create a new archive but fail to add any files to it, a 0 length file is left on the disk. It's your responsibility to delete such files (see LZH/LZHO for an example of this situation).

Example

```
var
  L : Lzh;
  Result : Word;
begin
  L.Create('MYLZH.LZH');
  if ArchiveStatus <> ecOk then begin
    WriteLn('Failed to create archive, error: ', ArchiveStatus);
    Halt;
  end;
  L.Compress('*.EXE');
  Result := L.GetLastError;
  if Result <> ecOk then
    WriteLn('Failed due to error ', Result);
  L.Done;
end;
```

Creates a new archive file named MYLZH.LZH and adds all files with the extension EXE to it.

See Also

UnLzh.Done

UnLzh.Init

Syntax

```
procedure CreateLzhFile(LzhName : PathStr);
```

Purpose

Create a new LZH archive file.

Description

This procedure creates a new LZH archive file. It creates and initializes a new LZH archive using the standard LZH format used by LHARC and LHA.

An extension of LZH is assumed if none is specified.

This procedure fails with `ecCannotCreate` if an archive with the specified name already exists. If any DOS I/O errors are encountered, the procedure fails and `ArchiveStatus` contains the DOS error code.

If you create a new archive but fail to add any files to it, a 0 length file is left on the disk. It's your responsibility to delete such files (see LZH/LZHO for an example of this situation).

Example

```
CreateLzhFile('MYLZH.LZH');
if ArchiveStatus <> ecOk then begin
  WriteLn('Failed to create archive, error: ', ArchiveStatus);
  Halt;
end;
CompressLzh('*.EXE');
if ArchiveStatus <> ecOk then
  WriteLn('Failed due to error ', ArchiveStatus mod 10000);
DoneLzhFile;
```

Creates a new archive file named MYLZH.LZH and adds all files with the extension EXE to it.

See Also

`DoneLzhFile`

`InitLzhFile`

Syntax

```
procedure LzhFileList.Delete(LNP : LzhNodePtr);
```

Purpose

Delete the specified node from the `LzhFileList`.

Description

This procedure deletes the specified node from the list and releases the memory associated with it. Note that the data associated with the node is permanently lost.

Example

```
LFL.Delete(LFL.lfHead);
```

Deletes the first node in the list.

See Also

`DeleteLzhFileListNode`

DeleteFileMaskList / DeleteFiles

Syntax

```
procedure DeleteFileMaskListLzh(var FML : FileMaskList);  
procedure Lzh.DeleteFileMaskList(var FML : FileMaskList);
```

Purpose

Remove all files that match any mask in the file mask list from an archive.

Description

This procedure removes files from an existing archive. The files to be removed are described by the specified FileMaskList.

Example

```
var  
  L : Lzh;  
  FML : FileMaskList  
  Result : Word;  
begin  
  FML.Init;  
  if not FML.Append('*.PAS') or  
    not FML.Append('*.EXE') then  
    {handle error} ;  
  L.Init('MYLZH.LZH');  
  if ArchiveStatus <> ecOk then  
    {handle error} ;  
  L.DeleteFileMaskList(FML);  
  Result := L.GetLastError;  
  if Result <> ecOk then  
    {handle errors} ;  
  L.Done;  
end.
```

Removes all files with extensions of PAS or EXE from MYLZH.LZH.

See Also

DeleteFiles

Syntax

```
procedure DeleteFilesLzh(Mask : PathStr);  
procedure Lzh.Delete(Mask : PathStr);
```

Purpose

Remove all files that match a file mask from an archive.

Description

This is a simplified version of DeleteFileMaskList. You may find this more convenient when all of the files to be deleted can be described by a single mask.

See DeleteFileMaskList for more information.

Syntax

```
procedure DeleteLzhFileListNode(var LFL : LzhFileList; LNP : LzhNodePtr);
```

Purpose

Delete the specified node from the LzhFileList.

Description

This procedure deletes the specified node from the LzhFileList and releases the memory associated with it. Note that the data associated with the node is permanently lost.

Example

```
DeleteLzhFileListNode(LFL, LFL.Head);
```

Deletes the first node in the list.

See Also

LzhFileList.Delete

Syntax

```
destructor LzhFileList.Done; virtual;
```

Purpose

Destroy an LzhFileList.

Description

This destructor destroys an LzhFileList, releasing all the memory allocated to the individual nodes in the list.

Example

See the example for Init.

See Also

DoneLzhFileList

Init

Syntax

```
destructor LzhNode.Done; virtual;
```

Purpose

Destroy a node.

Description

This destructor destroys the node. It is intended primarily for internal use by LzhFileList.Done.

See Also

LzhFileList.Done

DoneLzhFile / DoneLzhFileList

Syntax

procedure DoneLzhFile;

Purpose

Close the LZH file.

Description

This routine closes the LZH file opened by InitLzhFile.

Example

See the example for InitLzhFile.

See Also

InitLzhFile

Syntax

procedure DoneLzhFileList(var LFL : LzhFileList);

Purpose

Destroy an LzhFileList.

Description

This routine destroys an LzhFileList created by previous calls to InitLzhFileList and BuildLzhFileList, releasing all the memory allocated to the individual nodes in the list.

Example

See the example for InitLzhFileList.

See Also

BuildLzhFileList
LzhFileList.Done

InitLzhFileList

Syntax

```
procedure ExtractLzh(Mask : PathStr);  
procedure UnLzh.Extract(Mask : PathStr);
```

Purpose

Extract all files that match the specified file mask.

Description

This procedure extracts all files from the LZH file that match the specified Mask. Mask can be a specific file, or it can be a file mask.

Example

```
var  
  UL : UnLzh;  
begin  
  if not UL.Init('FOO.LZH') then  
    {... handle error ...};  
  UL.Extract('*.*');  
  if UL.GetLastError <> 0 then  
    {... handle error ...};  
  UL.Done;  
end.
```

This program extracts all files in FOO.LZH.

See Also

ExtractFileMaskList

Syntax

```
procedure ExtractFileMaskListLzh(var FML : FileMaskList);  
procedure UnLzh.ExtractFileMaskList(var FML : FileMaskList);
```

Purpose

Extract all files that match the file mask list.

Description

This routine extracts all files that match one or more of the file masks in the specified FileMaskList.

Examples

See the examples for InitLzhFile and UnLzh.Init.

See Also

ExtractLzhFileList
Archive.GetLastError

InitFileMaskList (APARCHIV)

ExtractLzhFileList / FreshenArchive

Syntax

```
procedure ExtractLzhFileList(var LFL : LzhFileList);  
procedure UnLzh.ExtractLzhFileList(var LFL : LzhFileList);
```

Purpose

Extract all files in the specified LzhFileList.

Description

This routine extracts all files in LFL, previously created by calling BuildLzhFileList.

Note that, in most cases, you should use ExtractFileMaskList instead of this routine.

ExtractLzhFileList is preferred only in cases where you need to use the list of files to be extracted for some other purpose (e.g., displaying a list of files to be extracted before they are actually extracted).

Examples

See the examples for InitLzhFileList and LzhFileList.Init.

See Also

BuildLzhFileList
Extract

ExtractFileMaskList
InitLzhFileList

Syntax

```
procedure FreshenArchiveLzh;  
procedure Lzh.FreshenArchive;
```

Purpose

Conditionally update files in the current archive.

Description

This procedure conditionally updates, or freshens, all files in the current LZH archive file. It processes the existing archive file sequentially. For each file in the archive, it looks for the same file in the current directory or the directory specified by the stored path name. If it finds such a file, it calls the OkToCompress function and passes it the LZH header of the compressed file and the name of the matching file. The OkToCompress function decides which version belongs in the archive. If a compressed file is not found on disk, the compressed file is transferred directly to the new archive.

If the OkToCompress function returns True, a new compressed image is created from the file on disk. If the OkToCompress function returns False, the existing compressed image is kept.

If no OkToCompress function was specified, all files are transferred to the new archive.

The supplied function DefOkToCompress compares the old and new file dates and returns True only if the new file date is later than the old file date.

Example

```
var  
  L : Lzh;  
begin  
  L.Init('MYLZH');  
  L.FreshenArchive;  
  L.Done;  
end;
```

A simple program to freshen all of the files in MYLZHLZH.

Syntax

```
constructor Lzh.Init(FName : PathStr);
```

Purpose

Open an LZH archive.

Description

This constructor initializes the archive and opens the input file FName. It does this by calling UnLzh.Init. See UnLzh.Init for more information.

Syntax

```
constructor LzhFileList.Init;
```

Purpose

Initialize an LzhFileList.

Description

This constructor initializes an LzhFileList in preparation for a call to UnLzh.BuildLzhFileList.

Example

```
var
  FML : FileMaskList;
  LFL : LzhFileList;
  UL : UnLzh;
begin
  FML.Init;
  LFL.Init;
  if not UL.Init('FOO.LZH') then (... handle error ...);
  UL.BuildLzhFileList(LFL, FML);
  if UL.GetLastError <> 0 then (... handle error ...);
  UL.ExtractLzhFileList(LFL);
  if UL.GetLastError <> 0 then (... handle error ...);
  UL.Done;
  LFL.Done;
  FML.Done;
end.
```

This program extracts all files in FOO.LZH. The examples for UnLzh.Init and UnLzh.Extract contain simpler versions of this program.

See Also

InitLzhFileList

Syntax

```
constructor LzhNode.Init(var LH : LzhHeader; FO : LongInt);
```

Purpose

Initialize node.

Description

This constructor initializes the node by storing the LH and FO parameters and setting the InNext pointer to nil. It is intended primarily for internal use by LzhFileList.Append.

See Also

LzhFileList.Append

Init / InitLzhFile

Syntax

constructor UnLzh.Init(FName : PathStr);

Purpose

Initialize the archive and open the input file.

Description

This constructor attempts to open the specified LZH file. It fails only if the file cannot be opened successfully or if the specified file is not a valid LZH file.

Example

```
var
  FML : FileMaskList;
  UL : UnLzh;
begin
  FML.Init;
  if not UL.Init('FOO.LZH') then
    {... handle error ...};
  UL.ExtractFileMaskList(FML);
  if UL.GetLastError <> 0 then
    {... handle error ...};
  UL.Done;
  FML.Done;
end.
```

This simplistic program extracts all files in FOO.LZH.

Syntax

procedure InitLzhFile(LzhName : PathStr);

Purpose

Open the LZH file.

Description

This routine opens the specified file and verifies that it is indeed an LZH file. If it isn't, ArchiveStatus is set to ecNotAnLzhFile.

The mode in which the file is opened depends on the current setting of Turbo Pascal's FileMode variable. If you want to open the file in read-only mode, set FileMode to \$20 before calling Init.

Example

```
var
  FML : FileMaskList;
begin
  InitFileMaskList(FML);
  InitLzhFile('FOO.LZH');
  if ArchiveStatus <> 0 then
    {... handle error ...};
  ExtractFileMaskList(FML);
  if ArchiveStatus <> 0 then
    {... handle error ...};
  DoneLzhFile;
  DoneFileMaskList(FML);
end.
```

This simplistic program extracts all files in FOO.LZH.

See Also

DoneLzhFile

UnLzh.Init

Syntax

```
procedure InitLzhFileList(var LFL : LzhFileList);
```

Purpose

Initialize an LzhFileList.

Description

Initializes an LzhFileList in preparation for a call to BuildLzhFileList.

Example

```
var
  FML : FileMaskList;
  LFL : LzhFileList;
begin
  InitFileMaskList(FML);
  InitLzhFileList(LFL);
  InitLzhFile('FOO.LZH');
  if ArchiveStatus <> 0 then
    {... handle error ...};
  BuildLzhFileList(LFL, FML);
  if ArchiveStatus <> 0 then
    {... handle error ...};
  ExtractLzhFileList(LFL);
  if ArchiveStatus <> 0 then
    {... handle error ...};
  DoneLzhFile;
  DoneLzhFileList(LFL);
  DoneFileMaskList(FML);
end.
```

This program extracts all files in FOO.LZH. A simpler version of this program is in the example for InitLzhFile.

See Also

BuildLzhFileList
DoneLzhFileList

ExtractLzhFileList
LzhFileList.Init

Syntax

```
procedure SetCompressionModeLzh(CM : CompressionMode);
procedure Lzh.SetCompressionMode(CM : CompressionMode);
```

Purpose

Set the compression mode.

Description

The Lzh compression routines are capable of two types of compression. The default compression mode is cmBestMethod. This uses the newest freezing method, which is used by LHA, or simply stores the file if the frozen file would be larger. Call SetCompressionMode to force the compression routines to always use the LHA freezing method (cmFrozen2), the older LHARC freezing method (cmFrozen1), or no compression method (cmStored). It can also be used to reset to the best method algorithm.

Example

```
SetCompressionMode(cmFrozen1);
Force the Lzh routines to use only the LHARC freezing method.
```

SetCompressSuccessFunc

Syntax

```
procedure SetCompressSuccessFuncLzh(CSF : CompressSuccessFunc);  
procedure Lzh.SetCompressSuccessFunc(CSF : CompressSuccessFunc);
```

Purpose

Specify a function to call after each file is compressed.

Description

Use this procedure to specify a CompressSuccess function. A CompressSuccess function is called after each file is compressed, giving you a chance to update your status display.

If using APLZH, the CompressSuccess function must be of the form:

```
($F+)  
function CompressSuccess(var LH : LzhHeader; ErrCode : Word) : Boolean;  
begin  
  ...  
end;
```

This function should return True if you want the compression process to continue.

If using OOLZH, the form should be:

```
($F+)  
function CompressSuccess(LP : LzhPtr; LH : LzhHeader) : Boolean;  
begin  
  ...  
end;
```

The OOP declaration also includes a pointer (LP) to the object that made the call. You can use this pointer to obtain the results of the compression (with LP^.GetLastError). Note that you can override the virtual method OkToCompress in lieu of calling SetOkToCompressFunc.

Both OOP and non-OOP declarations include LH, the LzhHeader of the file that was just compressed. The compression method can be found by examining LH.HeadID[4].

If the compress failed, then ErrCode (or LP^.GetLastError) contains an Async Professional error code describing the cause of the failure. Generally, you should abort the compression process to restore the original archive for all errors—do this by having your CompressSuccess function return False. It is possible, however, to have the compression process continue after some errors. For example, if the compressor can't find a file specified in the file mask, it calls the CompressSuccess function with an ErrCode of ecFileNotFound. You may wish to simply display a message describing the problem and return True to have the update process continue with the rest of the file mask list.

Note that even if your CompressSuccess function returns True after an error, the compressor may decide to abort and restore the original archive anyway. It does this for all Turbo Pascal errors between 100 and 163 since those errors indicate a fatal programming error or a hardware problem.

A default CompressSuccess function called DefCompressSuccessFunc is provided. This function updates the display, noting that the compression succeeded, and it displays a ratio of compressed size to original size. Like all other default hooks in APLZH/OOLZH, the end result is a display that looks very similar to that produced by the LHARC utility. DefCompressSuccessFunc returns False if ErrCode is non-zero, forcing the compression process to abort and the original archive to be restored.

Example

```
SetCompressSuccessFunc(DefCompressSuccessFunc);
```

This example uses the built-in function, DefCompressSuccessFunc, which displays the string "Frozen" and a ratio of compressed file size to original file size.

Syntax

```
procedure SetExtractSuccessFuncLzh(ESF : ExtractSuccessFunc);
procedure UnLzh.SetExtractSuccessFunc(ESF : ExtractSuccessFunc);
```

Purpose

Specify a function to call after a file is unarchived.

Description

This procedure specifies a function that will be called immediately after the extraction of a file, giving you an opportunity to report any errors that occurred during the extraction process.

If using APLZH, this function must be global and of the form:

```
{F+}
function ExtractSuccess(var LH : LzhHeader; FName : PathStr;
                        ErrCode : Word) : Boolean;

begin
  ...
end;
```

The function should return False if it wants to abort the extract operation, either because of the nature of the error (e.g., disk full) or because the user asked to abort it. ErrCode contains 0 if the extraction was successful, else an error code. LH is the header for the file that was being extracted, and FName is the name of the output file that was being created.

If using OOLZH:

```
{F+}
function ExtractSuccess(LNP : LzhNodePtr; FName : PathStr;
                        UP : UnLzhPtr) : Boolean;

begin
  ...
end;
```

The code for the error encountered, if any, can be obtained by calling UP^.GetLastError. The header for the file can be accessed as LNP^.LnLH.

Note that, when using OOLZH, you can override the virtual method ExtractSuccess in lieu of calling SetExtractSuccessFunc.

Example

```
SetExtractSuccessFunc(DefExtractSuccessFunc);
```

This example uses the provided DefExtractSuccessFunc, which reports and treats as non-fatal (i.e., not fatal to the entire extraction operation) the following errors: ecUnknownMethod, ecCannotCreate, and ecBadFileCRC. All other errors are considered fatal to the extraction operation.

SetOkToCompressFunc

Syntax

```
procedure SetOkToCompressFuncLzh(OKC : OkToCompressFunc);  
procedure Lzh.SetOkToCompressFunc(OKC : OkToCompressFunc);
```

Purpose

Specify a function to call before compressing a file.

Description

Use this function to specify an OkToCompress function—a function that is called before a file is compressed, giving you the chance to skip the compression.

If using APLZH, the OkToCompress function must be of the form:

```
{F+}  
function OkToCompress(NewFile : PathStr; LH : LzhHeader) : Boolean;  
begin  
  ...  
end;
```

The function should return True if NewFile should be compressed, False if NewFile should be skipped.

If using OOLZH, the form should be:

```
{F+}  
function OkToCompressFunc(LP : LzhPtr; NewFile : PathStr;  
                           LH : LzhHeader) : Boolean;  
begin  
  ...  
end;
```

When using OOLZH, you can override the virtual method OkToCompress in lieu of calling SetOkToCompressFunc.

NewFile has the complete path name of the file that is to be compressed. LH is the LzhHeader record of a matching file in the archive (if there is one). The OkToCompress function is your opportunity to approve the addition of a file to the archive or to approve the replacement of an existing file.

Your OkToCompress function should handle the following two modes of operation:

- 1) If LH.FName is blank, NewFile describes a new file to be compressed. You need to decide whether you want to add it to the archive. For example, you may want to exclude all .BAK files from the archive no matter what file masks are specified in the list. Note that LH.FName is the only field in the LH header that you can assume has been initialized. If LH.FName is blank, the rest of the record is *not* initialized. Your OkToCompress function should return True if NewFile is to be added to the archive and False if it should be skipped.
- 2) If LH.FName is not blank, NewFile and LH.FName are the same file and you need to decide whether to replace the file (compress NewFile into the new archive) or leave the file as-is (copy LH.FName from the old archive to the new archive). In this case, LH contains the actual file header from the archive. You can reference any field in the header to help you decide whether to replace the file. For example, you would probably want to replace the file if the date/time stamp of NewFile is later than the date/time stamp from the LH header.

Your OkToCompress function should return True if the file is to be replaced in the archive and False if the old file should be retained. Note that there is no way to skip an existing file—it is either copied or replaced. If you need to delete an existing file, make a separate call to DeleteFiles or DeleteFileMaskList.

Async Professional provides a default OkToCompress function called DefOkToCompressFunc. For the first case described above, this default function returns True. For the second case, it compares the date/time stamps of the two files and returns True if NewFile's date/time stamp is later than LH.FName's date/time stamp, and returns False otherwise.

If you supply your own OkToCompress function, it must be declared exactly as above and be both global and far (using {\$F+} or the far keyword). If you do not supply an OkToCompress function, new files are always added to the archive and matching files are always replaced.

Example

```
SetOkToCompressFunc(DefOkToCompressFunc);
```

This example uses the built-in function, DefOkToCompressFunc, which always returns True for new files and also returns True for existing files if the file on disk has a later date/time stamp.

SetOkToWriteFunc

Syntax

```
procedure SetOkToWriteFuncLzh(OKF : OkToWriteFunc);  
procedure UnLzh.SetOkToWriteFunc(OKF : OkToWriteFunc);
```

Purpose

Specify a function to call before extracting a file.

Description

Use this procedure to specify a function that will be called just before a file is extracted, giving you an opportunity to decide whether or not it should actually be extracted. The OkToWrite function can also change the name of the file to be written.

If using APLZH, the OkToWrite function must be global and of the form:

```
{ $F+ }  
function OkToWrite(var LH : LzhHeader; var FName : PathStr) : Boolean;  
begin  
    ...  
end;
```

The function should return False if the file should not be extracted (e.g., because it already exists). FName is the name of the output file that will be created if the file is extracted. The LH parameter gives you access to the date/time stamp for the file. See the OkToWrite function in LZHX.PAS for an example of how to use this information.

If using OOLZH:

```
{ $F+ }  
function OkToWrite(LNP : LzhNodePtr; var FName : PathStr;  
                  UP : UnLzhPtr) : Boolean;  
begin  
    ...  
end;
```

The header for the file can be accessed as LNP^.LnLH. OOLZH passes an LzhNodePtr here, rather than an LzhHeader, for consistency with OOZIP.

See the OkToWrite function in LZHXO.PAS for a good example.

Note that, when using OOLZH, you can override the virtual method OkToWrite in lieu of calling SetOkToWriteFunc.

Example

```
SetOkToWriteFunc(DefOkToWriteFunc);
```

This example uses the supplied DefOkToWrite function, which always returns True.

Syntax

```
procedure SetProgressWidth(Width : Word);  
procedure Lzh.SetProgressWidth(Width : Word);
```

Purpose

Set the display width used by DefShowProgress.

Description

This procedure modifies the display width assumed by DefShowProgressFunc. By default, DefShowProgress assumes a screen width of 80 columns. It uses this for calculating the number, and byte equivalent, of the dots that it draws for the progress status display. If you wish to limit this "dot display" to a smaller window, call SetProgressWidth and pass it the interior width of that window.

Note that DefShowProgressFunc uses standard output to draw its row of dots. In many cases this, coupled with use of SetProgressWidth, may be enough for you to create simple but visually appealing status displays. If you need more complex displays, you have the option of creating a completely new ShowProgress function.

Example

```
var  
  L : Lzh;  
begin  
  L.SetProgressWidth(60);  
  L.SetCompressSuccessFunc(DefCompressSuccessFunc);  
  Window(1, 1, 60, 25);  
  L.Compress(Mask);  
  ....
```

This example uses DefCompressSuccessFunc but limits the width to 60 characters. This is required because a 60 column window is enabled just before the call to Compress.

See Also

SetShowProgressFunc

SetShowMethodProc

Syntax

```
procedure SetShowMethodProcLzh(SMP : ShowMethodProc);  
procedure UnLzh.SetShowMethodProc(SMP : ShowMethodProc);
```

Purpose

Specify a procedure to display the file name and compression method.

Description

Use this procedure to specify a routine that will be called just before a file is compressed or extracted, giving you an opportunity to display its name and the compression method used.

If using APLZH, the ShowMethod procedure must be global and of the form:

```
{SF+}  
procedure ShowMethod(Method : Char; FName : PathStr);  
begin  
  ...  
end;
```

Method will be either cmcStored (no compression), cmcFrozen1, or cmcFrozen2. FName is the name of the output file that is created; it may or may not be a complete pathname.

If using OOLZH:

```
{SF+}  
procedure ShowMethod(LNP : LzhNodePtr; FName : PathStr; UP : UnLzhPtr);  
begin  
  ...  
end;
```

The compression method can be found by examining LNP^.lnLH.HeadID[4].

Note that, when using OOLZH, you can override the virtual method ShowMethod in lieu of calling SetShowMethodProc.

Example

```
SetShowMethodProc(DefShowMethodProc);
```

This example uses the supplied DefShowMethodProc, which writes either 'Extracting' (Method = cmcStored), 'Melting' (Method = cmcFrozen1 or cmcFrozen2), 'Freezing', or 'Deleting' followed by the filename passed to it (FName).

Syntax

```
procedure SetShowNameProcLzh(SNP : ShowNameProc);  
procedure UnLzh.SetShowNameProc(SNP : ShowNameProc);
```

Purpose

Specify a procedure to display the LZH file name.

Description

Use this procedure to specify a procedure that will be called when an LzhFileList is being built by BuildLzhFileList, giving you an opportunity to display the name of the LZH file. The call is made after it has been determined that the file is a valid LZH file, and before the scan for matching files begins.

If using APLZH, the ShowName procedure must be global and of the form:

```
{SF+}  
procedure ShowName(FName : PathStr);  
begin  
  ...  
end;
```

FName is the name of the LZH file being used to build the LzhFileList.

If using OOLZH:

```
{SF+}  
procedure ShowName(UP : UnLzhPtr);  
begin  
  ...  
end;
```

The name of the file can be obtained by calling UP^.GetFileName.

Note that, when using OOLZH, you can override the virtual method ShowName in lieu of calling SetShowNameProc.

Example

```
SetShowNameProc(DefShowNameProc);
```

This example uses the supplied DefShowNameProc, which simply writes 'Searching' followed by the filename passed to it (FName).

See Also

BuildLzhFileList

SetShowProgressFunc

Syntax

```
procedure SetShowProgressFuncLzh(SPF : ShowProgressFunc);  
procedure UnLzh.SetShowProgressFunc(SPF : ShowProgressFunc);
```

Purpose

Specify a function to call to show the progress of a compression or extraction.

Description

Use this procedure to specify a function that will be called periodically during the compression or extraction of a file, giving you an opportunity to indicate to the user that progress is being made. During compression this function is called after NextDisplayInc bytes of the source file are processed. You can adjust the frequency by modifying NextDisplayInc (default is 4K). During extraction the function is called each time the output buffer is flushed to disk. (This buffer is 4K in size and it cannot be resized.)

If using APLZH, the ShowProgress function must be global and of the form:

```
{SF+}  
function ShowProgress(BytesWritten, TotalBytes : LongInt) : Boolean;  
begin  
    ...  
end;
```

If using OOLZH:

```
{SF+}  
function ShowProgress(UP : UnLzhPtr;  
                     BytesWritten, TotalBytes : LongInt) : Boolean;  
begin  
    ...  
end;
```

Note that, when using OOLZH, you can override the virtual method ShowProgress in lieu of calling SetShowProgressFunc.

The ShowProgress function should return False if it wants to abort the compress or extract operation (e.g., because the user pressed ^Break or ^C). During compression, BytesWritten is the number of original input bytes compressed so far and TotalBytes is the size of the original file. During extraction, BytesWritten is the number of bytes written so far and TotalBytes is the (uncompressed) size of the file being extracted. In the final call to ShowProgress, BytesWritten may be slightly smaller than TotalBytes (never reaching 100% complete). The subsequent call to CompressSuccess or ExtractSuccess is your indication that all bytes were processed (100% complete).

The default routine, DefShowProgressFunc, works as follows during compression:

DefShowProgressFunc displays a row of dots (periods) on the first call (when BytesWritten is zero). Each dot is equal to 4096 bytes of the input file. So, a file of exactly 20000 bytes would be represented by 5 dots: $(20000 \div 4096) + 1$.

If you are using DefShowMethodFunc and an 80 column monitor, 63 columns are available for displaying dots. Hence, the largest file whose progress can be displayed in this manner is 258048 bytes (4096 bytes times 63 columns). If the input file size is larger than this, the DefShowProgress routine alters the scale of each dot such that the size of the file is exactly represented by a full row of 63 dots. It also updates the global variable NextDisplayInc to adjust the frequency at which the compressor calls DefShowProgressFunc.

On subsequent calls to DefShowProgressFunc, each dot is changed in sequence to the small letter 'o'. The resulting visual effect is similar to that produced by the LHARC utility.

Note that DefShowProgressFunc uses standard output to display its dots and assumes that you are working with a screen width of 80 columns. If you want to limit the output to a different width (e.g., if you are displaying the progress in a window) while still using DefShowProgressFunc, call SetProgressWidth and pass it the interior width of the window. See SetProgressWidth for more information.

DefShowProgressFunc works as follows during extraction:

DefShowProgressFunc displays dots (periods) as the file is extracted. Each dot is equal to 4096 bytes of the input file. So, a file exactly 20000 bytes would be represented by 5 dots: $(20000 \div 4096) + 1$.

Example

```
SetShowProgressFunc(DefShowProgressFunc);
```

This example uses the supplied DefShowProgressFunc, which works as described above.

SetTag

Syntax

procedure LzhNode.SetTag(On : Boolean);

Purpose

Tag/untag this node.

Description

This method tags or untags the file associated with this node. If the file is not tagged, it cannot be extracted from the LZH file. All files are initially tagged by LzhNode.Init.

D. ZIP Archives: APZIP/OOZIP

These two units provide facilities for working with ZIP archives. They allow you to archive files, list the files stored in an archive, and extract files from an archive. At this time, APZIP and OOZIP work only with ZIP files that can be processed successfully by PKZIP version 1.x.

The ZIP object hierarchy is:

```
Archive
├── UnZip
│   └── Zip
```

The Zip object, which compresses files to a ZIP archive, is derived from the UnZip object, which decompresses files from a ZIP archive. In this way, the Zip object can share much of the "boilerplate" code in the UnZip object responsible for reading and writing file headers, calculating CRCs, and so on. The only downside to this hierarchy is that it isn't possible to have a "compress only" object--your Zip objects will always include a few virtual methods that apply only to decompression.

Unlike the LZH compression routines, the ZIP routines use relatively little data segment space--less than 2K, in fact. However, the ZIP compression routines use almost 60K of heap space counting all compression and I/O buffers. This heap space is allocated at the start of all compression and extraction routines (e.g., CompressFileMaskList and ExtractFileMaskList) and is disposed before the routines return.

Note that APZIP and OOZIP work with self-extracting files created with ZIP2EXE, as well as with regular ZIP files. One thing they cannot do, however, is extract password-encrypted files stored in a ZIP archive (that is, files scrambled with PKZIP's -S option).

The remainder of this section discusses how to view a ZIP archive directory, how to extract files, how to compress files, the available user hooks, and the structure of a ZIP archive.

Viewing the Directory of a ZIP Archive

The following sample program, ZIPLIST, is analogous to the LZHLIST program presented in §11.C. ZIPLIST takes the name of a ZIP file to view as a command line parameter. If the file is opened successfully and is indeed a ZIP archive, it displays the names and the uncompressed sizes of all the files in the archive.

```
program ZipList; {ZIPLIST.PAS}
uses
  ApMisc, ApArchiv, ApZip;
var
  FML : FileMaskList;
  ZFL : ZipFileList;
  ZNP : ZipNodePtr;
begin
  {create an empty file mask list}
  InitFileMaskList(FML);

  {initialize ZIP file list}
  InitZipFileList(ZFL);

  {open the ZIP file}
  InitZipFile( ParamStr(1) );
  if ArchiveStatus <> 0 then begin
    Writeln('Error: ', StatusStr(ArchiveStatus));
    Halt;
  end;
end;
```

```

{enable user hooks}
SetShowCommentsProc(DefShowCommentsProc);

{construct list of files in archive}
BuildZipFileList(ZFL, FML);
if ArchiveStatus <> 0 then begin
  Writeln('Error: ', StatusStr(ArchiveStatus));
  Halt;
end;

{display names and sizes of all files in archive}
ZNP := ZFL.Head;
while ZNP <> nil do begin
  with ZNP^, CDH do
    Writeln(OrigSize:7, ' ', FName);
  ZNP := ZNP^.Next;
end;

{close the ZIP file}
DoneZipFile;

{dispose of data structures}
DoneZipFileList(ZFL);
DoneFileMaskList(FML);
end.

```

If you compare this program to LZHLIST, you'll see that they are largely identical. Other than some obvious name changes, there aren't many differences between them. A couple of differences are worth discussing, however.

First, there's no ShowName hook for ZIP files, but rather a ShowComments hook. A ShowComments routine serves the same purpose, but it may also be given an opportunity to display a comment placed in the file by the user concerning the archive as a whole. (This archive comment is read only if arReadArcComments option is enabled.)

Second, the structure of a ZipNode is quite a bit different than that of an LzhNode (the following declaration is from APZIP):

```

ZipNode =
  record
    Next   : ZipNodePtr;
    CDH    : CentralDirHead;
    FName  : PathStr;
    CP     : CommentPtr;
    EP     : ExtraPtr;
    Tagged : Boolean;
  end;

```

The CDH field is comparable to the LH field in an LzhNode. It's where most of the data about the compressed file is stored. Note that FName, the name of the file, is stored in the ZipNode, not in the header as it is in an LZH file. (The reason for this difference is made clear in "Structure of a ZIP File" later in this section.) The CP field has no equivalent in an LzhNode. If not nil, this field points to an array of characters containing comments about this particular file. (Such comments are read into memory only if the arReadFileComments option is enabled.) The length of the array is found in CDH.CommentLength. EP is a pointer to an extra field that is not currently used.

The ZIP file's equivalent of the LzhHeader is the CentralDirHead:

```

CentralDirHead =
record
  VersionMade : Word;           {version created by}
  VersionNeeded : Word;        {version needed to extract}
  BitFlag : Word;              {general purpose bit flag}
  Method : Word;               {compression method: 0..6}
  Time : Word;                 {time file was last modified}
  Date : Word;                 {date file was last modified}
  Crc : LongInt;               {32-bit CRC}
  NewSize : LongInt;           {compressed size of file}
  OrigSize : LongInt;          {uncompressed size of file}
  NameLength : Word;           {length of filename}
  ExtraLength : Word;          {length of extra field}
  CommentLength : Word;        {length of comment for file}
  DiskNumberStart : Word;      {number of disk on which file begins}
  InternalAttrs : Word;        {internal file attributes, low bit = Text}
  ExternalAttrs : LongInt;     {external file attributes, DOS dir attr}
  LocalHeaderOfs : LongInt;     {relative offset of local header for file}
end;

```

This data structure contains the same kinds of information found in an LzhHeader: the size of the file before and after compression, the date/time stamp for the file, the compression method used, etc. Like LZHLIST, this simple example program displays only the name of the file and its uncompressed size, but a more sophisticated listing program would display most of the information in the CentralDirHead. See ZIPV.PAS for an example of how to interpret and format the various fields in the record. Additional information on this data structure can be found in the APPNOTE.TXT file distributed with PKZIP.

Here again, the object-oriented version of the program is strikingly similar to the procedural version:

```

program ZipList0; {ZIPLIST0.PAS}
uses
  Apmisc, OoArchiv, OoZip;
var
  UZ : UnZip;
  FML : FileMaskList;
  ZFL : ZipFileList;
  ZNP : ZipNodePtr;
begin
  {create an empty file mask list}
  FML.Init;

  {initialize ZIP file list}
  ZFL.Init;

  {open the ZIP file}
  if not UZ.Init( ParamStr(1) ) then begin
    Writeln('Error: ', StatusStr(ArchiveStatus));
    Halt;
  end;

  {enable user hooks}
  UZ.SetShowCommentsProc(DefShowCommentsProc);

```

```

{construct list of files in archive}
UZ.BuildZipFileList(ZFL, FML);
if UZ.GetLastError <> 0 then begin
  Writeln('Error: ', StatusStr(UZ.GetLastError));
  Halt;
end;

{display names and sizes of all files in archive}
ZNP := ZFL.zfHead;
while ZNP <> nil do begin
  with ZNP^, znCDH do
    Writeln(OrigSize:7, ' ', znFName);
  ZNP := ZNP^.znNext;
end;

{close the ZIP file}
UZ.Done;

{dispose of data structures}
ZFL.Done;
FML.Done;
end.

```

The only notable differences between this program and the previous one are 1) you use the `GetLastError` method rather than `ArchiveStatus` to check for errors, and 2) the names of data fields in `ZipNodes` and `ZipFileLists` are preceded by prefixes, 'zf-' for `ZipFileList` and 'zn-' for `ZipNode`. The declaration for the `CentralDirHead` type in `OOZIP` is identical to that in `APZIP`.

Extracting Files from a ZIP Archive

The process of extracting files from a ZIP archive is largely identical to that used for viewing its directory, as the following program illustrates:

```

program ZipExt; {ZIPEXT.PAS}
uses
  ApMisc, ApArchiv, ApZip;
var
  FML : FileMaskList;
begin
  {create an empty file mask list}
  InitFileMaskList(FML);

  {open the ZIP file}
  InitZipFile( ParamStr(1) );
  if ArchiveStatus <> 0 then begin
    Writeln('Error: ', StatusStr(ArchiveStatus));
    Halt;
  end;

  {enable user hooks}
  SetShowCommentsProc(DefShowCommentsProc);
  SetShowMethodProc(DefShowMethodProc);
  SetExtractSuccessFunc(DefExtractSuccessFunc);
  SetShowProgressFunc(DefShowProgressFunc);

  {extract all files in archive}
  ExtractFileMaskList(FML);
  if ArchiveStatus <> 0 then begin
    Writeln('Error: ', StatusStr(ArchiveStatus));
    Halt;
  end;
end;

```

```

    {close the ZIP file}
    DoneZipFile;

    {dispose of data structures}
    DoneFileMaskList(FML);
end.

```

With the exception of one line, ZIPEXT can be created from LZHEXT simply by replacing all instances of 'Lzh' with 'Zip'. The lone exception is the call to SetShowCommentsProc (discussed in the previous section), which is APZIP's equivalent of APLZH's SetShowNameProc.

The object-oriented version of the program is as follows:

```

program ZipExt0; {ZIPEXT0.PAS}
uses
    ApMisc, OoArchiv, OoZip;
var
    UZ : UnZip;
    FML : FileMaskList;
begin
    {create an empty file mask list}
    FML.Init;

    {open the ZIP file}
    if not UZ.Init( ParamStr(1) ) then begin
        Writeln('Error: ', StatusStr(ArchiveStatus));
        Halt;
    end;

    {enable user hooks}
    UZ.SetShowCommentsProc(DefShowCommentsProc);
    UZ.SetShowMethodProc(DefShowMethodProc);
    UZ.SetExtractSuccessFunc(DefExtractSuccessFunc);
    UZ.SetShowProgressFunc(DefShowProgressFunc);

    {extract all files in archive}
    UZ.ExtractFileMaskList(FML);
    if UZ.GetLastError <> 0 then begin
        Writeln('Error: ', StatusStr(UZ.GetLastError));
        Halt;
    end;

    {close the ZIP file}
    UZ.Done;

    {dispose of data structures}
    FML.Done;
end.

```

As with LZHEXT0, ZIPEXT0 could be simplified further by getting rid of the FileMaskList and calling

```
UZ.Extract('*.');
```

rather than UZ.ExtractFileMaskList.

Selecting Files to be Extracted

The means of selecting files to be extracted are basically the same for ZIP files as they are for LZH files:

1. Create a FileMaskList and pass it to ExtractFileMaskList.
2. Use BuildZipFileList to construct a list of files in the archive and use DeleteZipFileListNode (APZIP) or ZipFileList.Delete (OOZIP) to remove specific nodes from the list before calling ExtractZipFileList.
3. Build a ZipFileList with BuildZipFileList and tag/untag nodes as desired before calling ExtractFileList. A given node can be untagged like so:

APZIP	OOZIP
<pre>var ZNP : ZipNodePtr; ... ZNP^.Tagged := False;</pre>	<pre>var ZNP : ZipNodePtr; ... ZNP^.znTagged := False;</pre>

Updating a ZIP Archive

APZIP and OOOZIP support three types of compression/update routines:

Compress - new files are added and specified old files are updated in an existing or new archive file.

Delete - specified files are removed from an existing archive file.

Freshen - every file in an archive is compared to its unarchived version on disk and is updated in the archive if the unarchived version has a later date/time stamp.

When updating an archive, a new archive file is created and files are copied from the old archive, or added new, until all files are processed. The original archive is renamed to an extension of '!!!' and a new output file is created with the name of the original archive. Files in the old archive are copied to the new archive and new files are compressed into the new archive as required.

Should the update operation fail at any point, the output file is deleted and the '!!!' file is renamed back to its original name. When all files are processed, the new output file is completely up to date and the old '!!!' file is deleted.

Obviously, the update process fails if there is not enough disk space to hold both the old archive file and the new archive file. Since there's no foolproof way to predict how big the resulting archive will be, your program must be prepared to deal with ecDiskFull errors after calling an update routine. The original archive file is restored automatically, but your application has to decide how to proceed when it receives this error.

All APZIP/OOOZIP compression and update routines preserve existing data in a ZIP file, including files stored in unknown compression formats, file comments, extra data, and ZIP comments. For example, if a compressed file has a comment or extra data associated with it, the comment and extra data are also copied to the new archive file. However, you also have the opportunity to modify file and archive comments through facilities that will be described later.

Just as you build a list of file masks when extracting files, you also build a list of file masks when compressing or deleting files. Here's a bare-bones example of a non-OOP program that shows this:

```
program ExZip; {EXZIP.PAS}  
uses  
  Dos, ApMisc, ApArchiv, ApZip;  
var  
  FML : FileMaskList;
```

```

begin
  {Initialize the file mask list and add masks}
  InitFileMaskList(FML);
  if not AppendFileMask('EXZIP.PAS', FML) or
    not AppendFileMask('EXZIP.EXE', FML) then begin
    WriteLn('Insufficient memory!');
    Halt;
  end;

  {Create a new ZIP file}
  CreateZipFile('EXZIP.ZIP');
  if ArchiveStatus <> ecOk then begin
    WriteLn('Failed to create archive, error: ', ArchiveStatus);
    Halt;
  end;

  {Set options}
  SetShowMethodProc(DefShowMethodProc);
  SetShowProgressFunc(DefShowProgressFunc);
  SetShowCommentsProc(DefShowCommentsProc);

  {Set compressing-only options}
  SetOkToCompressFunc(DefOkToCompressFunc);
  SetCompressSuccessFunc(DefCompressSuccessFunc);

  {Add the files to the archive}
  CompressFileMaskList(FML);

  {Report errors}
  if ArchiveStatus <> ecOk then
    WriteLn('Failed due to error ', ArchiveStatus);

  {Clean up}
  DoneFileMaskList(FML);
  DoneZipFile;
end.

```

And here's the OOP version of the same program:

```

program ExZipO; {EXZIPO.PAS}
uses
  Dos, ApMisc, OoArchiv, OoZip;
var
  FML : FileMaskList;
  Z : Zip;
  Result : Word;
begin
  {Instantiate the file mask list and add masks}
  FML.Init;
  if not FML.Append('EXZIPO.PAS') or
    not FML.Append('EXZIPO.EXE') then begin
    WriteLn('Insufficient memory!');
    Halt(1);
  end;
end;

```

```

(Create ZIP file)
Z.Create('EXZIPO.ZIP');
if ArchiveStatus <> ecOk then begin
    WriteLn('Failed to create archive, error: ', ArchiveStatus);
    Halt;
end;

(Set standard options)
Z.SetShowMethodProc(DefShowMethodProc);
Z.SetShowProgressFunc(DefShowProgressFunc);
Z.SetShowCommentsProc(DefShowCommentsProc);

(Set compressing-only options)
Z.SetOkToCompressFunc(DefOkToCompressFunc);
Z.SetCompressSuccessFunc(DefCompressSuccessFunc);

(Add the files to the archive)
Z.CompressFileMaskList(FML);

(Report errors)
Result := Z.GetLastError;
if Result <> ecOk then
    WriteLn('Failed due to error ', Result);

(Clean up)
FML.Done;
Z.Done;
end.

```

The first step in both of these programs is to build the list of files to be added to the archive. In each case just the .PAS and .EXE files of the example program are added (i.e., EXZIP.PAS and EXZIP.EXE). If the file masks are added successfully, a new archive file is created. Note that this program fails if the archive file already exists. In a more sophisticated program you would probably want to handle this situation more gracefully (see ZIP and ZIPO for such an example).

After creating the archive file, the various user hooks to the Async Professional built-in functions are set. The first three user hooks are used above in file extraction. The last two user hooks are specific to compressing files and are described in more detail below. For now, it's enough to know that the OkToCompress function is called to verify that it's OK to compress each matching file. The CompressSuccess function is called after the file is compressed so you can update your status display with the compression results.

Next, the example program calls CompressFileMaskList to compress all files that match the masks in FML. In this case, FML contains exact file names rather than wildcard masks, but it works the same in either case. Files that match the masks are compressed into the archive.

The files are compressed using either the shrinking or imploding compression method, whichever produces the smallest compressed file. To force shrinking or imploding, use SetCompressionMode (see the description later in this section).

If any errors occur during the compression process, ArchiveStatus (or Z.GetLastError) will contain a code describing the cause of the failure.

Finally, the file mask list is disposed and the archive file is closed by a call to DoneZipFile (or Z.Done).

APZIP/OOZIP User Hooks

The user hooks available in APZIP/OOZIP are basically the same as those in APLZH/OOLZH. These eight hooks allow you to designate user-written functions that are called at specific points during the processing of a ZIP file.

An `OkToWrite` function is called before a file is extracted to give you an opportunity to prevent the operation (e.g., because the file already exists and either the existing file is newer than the archived one or the user doesn't want it overwritten). An `OkToWrite` function can also change the name of the file to be written (e.g., in cases where the file already exists, and you want to extract it under another name). The demonstration programs `ZIPX` and `ZIPXO` contain sophisticated examples of `OkToWrite` functions. See `SetOkToWriteFunc` for details.

A `ShowMethod` procedure is called just before a file is compressed or extracted, giving you an opportunity to display the name of the file and the compression method used on it. See `SetShowMethodProc` for details. Despite the name, the real purpose of this routine during compression is to display the current file name. If you want to display the compression method, you'll need to do it in the `CompressSuccess` function because the compression method isn't known until the compression is finished. The default routine, `DefShowMethodProc`, displays the file name on the current line and the string 'Freezing : ' on the line below it. When deleting files, `DefShowMethodProc` displays 'Deleting : ' and the filename on the same line.

A `ShowProgress` function is called regularly during the compression and extraction processes. This hook gives you an opportunity to 1) indicate to the user that progress is being made and 2) abort the operation if desired. The default routine, `DefShowProgressFunc`, displays dots ('.') during extraction and o's ('o') during compression. See `SetShowProgressFunc` for details.

An `ExtractSuccess` function is called after attempting to extract a file. This hook gives you an opportunity to display an error message if an error occurred, and to abort the extraction operation if desired. See `SetExtractSuccessFunc` for details. This function is not called when compressing files.

The `OkToCompress` hook is a boolean function that is called whenever an update routine finds a file that might need to be added to an archive or updated. Your `OkToCompress` function must determine if the specified file should be added to the archive, or if an existing file should be replaced in the archive. See `SetOkToCompressFunc` for details.

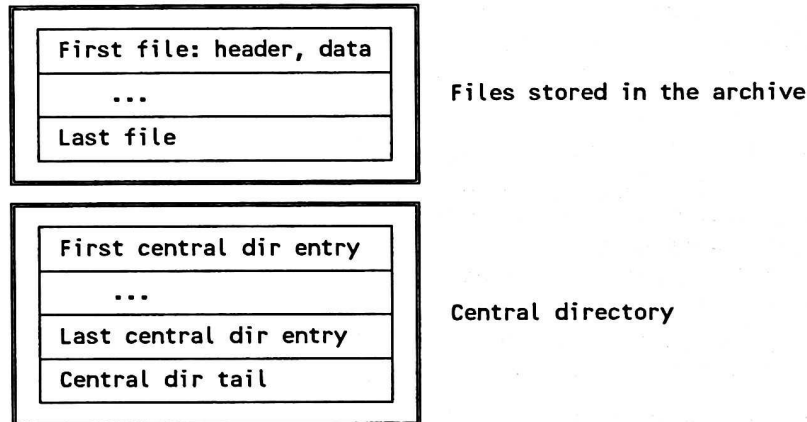
The `CompressSuccess` function is very similar in form and function to the `ExtractSuccess` function described above. This function is called by all update routines (compress or freshen) after each attempt to compress a file. Generally, the role of this function is simply to announce that the compress process is over and, perhaps, display a compression ratio or other information. See `SetCompressSuccessFunc` for details.

A `FileComment` function is called after each file is compressed, or for each file that meets the file mask list supplied to `UpdateCommentsFileMaskList`, to get a new file comment. See `SetFileCommentFunc` for details.

A `ShowComments` function is called at the beginning of any compression or extraction function. It allows you to display the name of the ZIP file and any comments concerning the ZIP file as a whole. The default routine, `DefShowCommentsProc`, displays 'Updating' (for compression) or 'Searching' (for extraction), followed by the file name and the comment, if any. See `SetShowCommentsProc` for details.

Structure of a ZIP File

A ZIP archive is divided into two major sections. The first section is similar to an LZH file: a series of two-part entries, with a header followed by the compressed data for the file. The second section is called the central directory. Each file stored in the archive has a corresponding entry in the central directory, which concludes with a variable-length data structure containing information about the archive as a whole. The following shows the structure of a ZIP archive:



The first step in processing a ZIP file is to search backwards from the end of the file (see the source for `InitZipFile`) for the 32-bit signature (`$06054B50`) that marks the "tail" of the central directory. You can then read in the following data structure:

```
CentralDirTail = record
  {... signature ...}           {= $06054B50}
  DiskNumber : Word;            {disk number for this disk}
  CentralDirDisk : Word;         {number of disk with start of central dir}
  EntriesThisDisk : Word;        {total entries in directory on this disk}
  TotalEntries : Word;           {total entries in central directory}
  CentralDirSize : LongInt;      {size of the central directory}
  CentralDirOfs : LongInt;       {offset of start of central dir with respect
                                to starting disk number}
  CommentLength : Word;         {length of ZIP file comment}
  {... ZIP file comment (variable size) ...}
end;
```

The ZIP file comment (a simple array of characters) is optional. If `CommentLength` is 0, there isn't one.

The `CentralDirOfs` field of the `CentralDirTail` points to the first entry in the central directory, which also begins with a 32-bit signature:

```
CentralDirHead = record
  {... signature ...}           {= $02014B50}
  VersionMade : Word;           {version created by}
  VersionNeeded : Word;         {version needed to extract}
  BitFlag : Word;               {general purpose bit flag}
  Method : Word;                {compression method: 0..6}
  Time : Word;                  {time file was last modified}
  Date : Word;                  {date file was last modified}
  Crc : LongInt;                {32-bit CRC}
  NewSize : LongInt;            {compressed size of file}
  OrigSize : LongInt;           {uncompressed size of file}
```

```

NameLength : Word;           {length of filename}
ExtraLength : Word;          {length of extra field}
CommentLength : Word;        {length of comment for file}
DiskNumberStart : Word;      {number of disk on which file begins}
InternalAttrs : Word;        {internal file attributes, low bit = Text}
ExternalAttrs : LongInt;     {external file attributes, DOS dir attr}
LocalHeaderOfs : LongInt;     {relative offset of local header for file}
{... filename (variable size) ...}
{... extra field (variable size) ...}
{... file comment (variable size) ...}
end;

```

This fixed-length data structure is followed by up to 3 variable-size fields: the filename (NameLength bytes), a block reserved for extra data (ExtraLength bytes), and a comment concerning the file (CommentLength bytes).

These entries in the central directory give the names of the files in the archive and their locations. By seeking to the offset given in the LocalHeaderOfs field, you can find the "local header" for the file, which repeats much of the information found in the CentralDirHead:

```

LocalHeader = record
  {... signature ...}          {= $04034B50}
  VersionNeeded : Word;        {version needed to extract}
  BitFlag : Word;              {general purpose bit flag}
  Method : Word;               {compression method: 0-6}
  Time : Word;                 {time file was last modified}
  Date : Word;                 {date file was last modified}
  Crc : LongInt;               {32-bit CRC}
  NewSize : LongInt;           {compressed size of file}
  OrigSize : LongInt;          {uncompressed size of file}
  NameLength : Word;           {length of filename}
  ExtraLength : Word;          {length of extra field}
  {... filename (variable size) ...}
  {... extra field (variable size) ...}
end;

```

The local header is followed by the filename (NameLength bytes), a block reserved for extra data (ExtraLength bytes), and then the compressed data for the file. For additional information concerning the format of a ZIP file, see APPNOTE.TXT (distributed with PKZIP) and study the source for APZIP.

Shared Declarations

Constants

```

cmcStored    = 0; {stored (no compression)}
cmcShrunk    = 1; {shrunk}
cmcReduced1  = 2; {reduced - factor of 1}
cmcReduced2  = 3; {reduced - factor of 2}
cmcReduced3  = 4; {reduced - factor of 3}
cmcReduced4  = 5; {reduced - factor of 4}
cmcImploded  = 6; {imploded}

```

The possible compression method codes.

Types

```

CentralDirHead =
  record
    {... signature ...}          {a LongInt, $02014B50}
    VersionMade : Word;          {version created by}

```

```

VersionNeeded : Word;      {version needed to extract}
BitFlag : Word;            {general purpose bit flag}
Method : Word;             {compression method: 0..6}
Time : Word;               {time file was last modified}
Date : Word;               {date file was last modified}
Crc : LongInt;             {32-bit CRC}
NewSize : LongInt;         {compressed size of file}
OrigSize : LongInt;        {uncompressed size of file}
NameLength : Word;         {length of filename}
ExtraLength : Word;        {length of extra field}
CommentLength : Word;      {length of comment for file}
DiskNumberStart : Word;    {number of disk on which file begins}
InternalAttrs : Word;      {internal file attributes, low bit = Text}
ExternalAttrs : LongInt;   {external file attributes, DOS dir attr}
LocalHeaderOfs : LongInt;  {relative offset of local header for file}
{... filename (variable size) ...}
{... extra field (variable size) ...}
{... file comment (variable size) ...}
end;

```

Entry for an individual file in the central directory of a ZIP file.

```

CentralDirTail =
record
  {... signature ...}      {a LongInt, $06054B50}
  DiskNumber : Word;        {disk number for this disk}
  CentralDirDisk : Word;    {number of disk with start of central dir}
  EntriesThisDisk : Word;   {total entries in directory on this disk}
  TotalEntries : Word;      {total entries in central directory}
  CentralDirSize : LongInt; {size of the central directory}
  CentralDirOfs : LongInt;  {offset of start of central dir with respect
                             to starting disk number}
  CommentLength : Word;     {length of ZIP file comment}
  {... zip file comment (variable size) ...}
end;

```

Record that marks the end of a ZIP file's central directory.

```

CommentPtr = ^ZipComment;
ZipComment = array[1..65521] of Char;

```

A comment in a ZIP file. Note that there is no length byte or terminating character (such as a #0). The length of the comment is stored separately.

```

LocalHeader =
record
  {... signature ...}      {a LongInt, $04034B50}
  VersionNeeded : Word;    {version needed to extract}
  BitFlag : Word;          {general purpose bit flag}
  Method : Word;           {compression method: 0-6}
  Time : Word;             {time file was last modified}
  Date : Word;             {date ""}
  Crc : LongInt;           {32-bit CRC}
  NewSize : LongInt;       {compressed size of file}
  OrigSize : LongInt;      {uncompressed size of file}
  NameLength : Word;       {length of filename}
  ExtraLength : Word;       {length of extra field}
  {... filename (variable size) ...}
  {... extra field (variable size) ...}
  {... compressed data ...}
end;

```

A header that precedes a compressed file in a ZIP archive.

CompressionMode = (cmBestMethod, cmStore, cmShrink, cmImplode);
Used by SetCompressionMode to select the ZIP compression type.

APZIP Declarations

Types

```
CompressSuccessFunc = function(var CDH : CentralDirHead; FName : PathStr;  
                               ErrCode : Word) : Boolean;
```

A function of this type is called after each file is compressed, giving you a chance to update your status display.

```
ExtractSuccessFunc = function(var CDH : CentralDirHead;  
                              FName : PathStr; ErrCode : Word) : Boolean;
```

A function called after a file is extracted from an archive. ErrCode contains 0 if the extraction was successful, else an error code. CDH is the header for the file being extracted, and FName is the name of the output file that was created. The function should return True if it wants the extraction operation to continue.

```
FileCommentFunc = function(var CDH : CentralDirHead; FName : PathStr;  
                           var CP : CommentPtr; var Len : Word) : Boolean;
```

A function of this type is called after each file is compressed, or for each file that meets the file mask list supplied to UpdateCommentsFileMaskList, to get a new file comment.

```
OkToCompressFunc = function(NewFile, OldFile : PathStr;  
                           var CDH : CentralDirHead) : Boolean;
```

A function of this type is called before each file is compressed, giving you the chance to skip the compression.

```
OkToWriteFunc = function(var CDH : CentralDirHead;  
                        var FName : PathStr) : Boolean;
```

A function of this type is called to determine whether or not the specified file (FName) should be extracted from the ZIP archive. This function can also change the name of the file to be written. Note that the call is made after any file mask matching has been done.

```
ShowCommentsProc = procedure(FName : PathStr; CP : CommentPtr; CLen : Word);
```

A procedure of this type is called when a ZipFileList is being built, giving you an opportunity to display the name of the ZIP file and the associated file comment, if any.

```
ShowMethodProc = procedure(Method : Byte; FName : PathStr);
```

A procedure of this type is called just before the specified file (FName) is compressed or extracted, giving you an opportunity to display its name and (during extraction) the compression method used (cmcStored through cmcImploded).

```
ShowProgressFunc = function(BytesWritten, TotalBytes : LongInt) : Boolean;
```

A function of this type is called periodically during the compression or extraction of a file, giving you an opportunity to indicate to the user that progress is being made. It should return False if it wants to abort the extract operation.

```
ZipFileList =  
  record  
    Head, Tail : ZipNodePtr;  
    Count : Word;  
  end;
```

A singly-linked list of files in a ZIP file, generally created by calling BuildZipFileList. Head points to the first node in the list, Tail to the last. Count is the number of nodes in the list.

```
ZipNodePtr = ^ZipNode;  
ZipNode =  
  record  
    Next : ZipNodePtr;  
    CDH : CentralDirHead;
```

```

FName : PathStr;
CP    : CommentPtr;
EP    : ExtraPtr;
Tagged : Boolean;
end;

```

A single node in a ZipFileList. FName is the name of the compressed file described by the CentralDirHead. CP is the comment associated with the file, if any. The length of the comment is CDH.CommentLength. Next points to the next node in the list. EP is a pointer to an extra field that is not currently used. Tagged may be set to False to prevent the file from being extracted.

Variables

```
Crc : LongInt;
```

Running CRC counter. This variable can be used by an ExtractSuccess function in the event of an ecBadFileCRC error to determine what the computed CRC value was. (CDH.CRC would contain the correct CRC value.)

OOZIP Declarations

Types

```
CompressSuccessFunc = function(ZNode : ZipNodePtr; ZP : ZipPtr) : Boolean;
```

A function of this type is called after each file is compressed, giving you a chance to update your status display.

```
ExtractSuccessFunc = function(ZNP : ZipNodePtr; FName : PathStr;
                             UP : UnZipPtr) : Boolean;
```

A function called after a file has been extracted from an archive. ZNP points the ZipNode corresponding to the file being extracted, and FName is the name of the output file that was being created. The success of the operation can be determined by calling UP^.GetLastError. The function should return True if it wants the extraction operation to continue.

```
FileCommentFunc = function(ZNode : ZipNodePtr; var CP : CommentPtr;
                           var Len : Word; ZP : ZipPtr) : Boolean;
```

A function of this type is called after each file is compressed, or for each file that meets the file mask list supplied to UpdateCommentsFileMaskList, to get a new file comment.

```
OkToCompressFunc = function(NewFile, OldFile: PathStr;
                            var CDH : CentralDirHead; ZP : ZipPtr) : Boolean;
```

A function of this type is called before each file is compressed, giving you the chance to skip the compression.

```
OkToWriteFunc = function(ZNP : ZipNodePtr; var FName : PathStr;
                         UP : UnZipPtr) : Boolean;
```

A function of this type is called to determine whether or not the specified file (FName) should be extracted from the ZIP archive. This function can also change the name of the file to be written. Note that the call is made after any file mask matching has been done.

```
ShowCommentsProc = procedure(CP : CommentPtr; CLen : Word; UP : UnZipPtr);
```

A procedure of this type is called when a ZipFileList is built, giving you an opportunity to display the name of the ZIP file and the associated file comment, if any.

```
ShowMethodProc = procedure(ZNP : ZipNodePtr; FName : PathStr; UP : UnZipPtr);
```

A procedure of this type is called just before the specified file (FName) is compressed or extracted, giving you an opportunity to display its name and (during extraction) the compression method used (cmcStored..cmclmplied).

```
ShowProgressFunc = function(UP : UnZipPtr;
                             BytesWritten, TotalBytes : LongInt) : Boolean;
```

A function of this type is called periodically during compression or extraction of a file, giving you an opportunity to indicate to the user that progress is being made. It should return False if it wants to abort the extract operation.

```

ZipPtr = ^Zip;
Zip =
  object(UnZip)
  ...
end;

```

An object used for compressing files into a ZIP archive.

```

UnZipPtr = ^UnZip;
UnZip =
  object(Archive)
    uzCRC : LongInt; {running CRC counter}
  ...
end;

```

An object used for extracting information and/or compressed files from a ZIP archive. uzCRC is a running CRC counter, used when extracting a file. This variable can be used by an ExtractSuccess function in the event of an ecBadFileCRC error to determine what the computed CRC value was.

```

ZipFileListPtr = ^ZipFileList;
ZipFileList =
  object
    zfHead : ZipNodePtr;
    zfTail : ZipNodePtr;
    zfCount : Word;
  ...
end;

```

A singly-linked list of files in a ZIP file, generally created by calling UnZip.BuildZipFileList. zfHead points to the first node in the list, zfTail to the last. zfCount is the number of nodes in the list.

```

ZipNodePtr = ^ZipNode;
ZipNode =
  object
    znNext : ZipNodePtr;
    znCDH : CentralDirHead;
    znFName : PathStr;
    znCP : CommentPtr;
    znEP : ExtraPtr;
    znTagged : Boolean;
  ...
end;

```

A single node in a ZipFileList. znFName is the name of the compressed file described by the CentralDirHead. znCP is the comment associated with the file, if any. The length of the comment is znCDH.CommentLength. znNext points to the next node in the list. znEP is a pointer to an extra field that is not currently used. znTagged can be set to False to prevent the file from being extracted.

Procedures and Functions

```

function Zip.CompressSuccess(ZNode : ZipNodePtr) : Boolean; virtual;
  {-Called after the file specified by ZNode has been compressed}

function UnZip.ExtractSuccess(ZNP : ZipNodePtr;
                             FName : PathStr) : Boolean; virtual;
  {-Called after the file specified by ZNP has been extracted}

function Zip.OkToCompress(NewFile, OldFile: PathStr;
                          var CDH : CentralDirHead) : Boolean; virtual;
  {-Returns True if OK to compress NewFile}

function UnZip.OkToWrite(ZNP : ZipNodePtr;
                        var FName : PathStr) : Boolean; virtual;
  {-Returns True if OK to write the file specified by ZNP}

```

```

function Zip.SetFileComment(ZNode : ZipNodePtr; var CP : CommentPtr;
                           var Len : Word) : Boolean; virtual;
    {-Called after compression to add a file comment}
procedure UnZip.ShowComments(CP : CommentPtr; CLen : Word); virtual;
    {-Called to display name of zip file and zip file comments}
procedure UnZip.ShowMethod(ZNP : ZipNodePtr; FName : PathStr); virtual;
    {-Called to display name of file being processed and compression method}
function UnZip.ShowProgress(BytesWritten, TotalBytes : LongInt)
                           : Boolean; virtual;
    {-Called to show progress during file updates}

```

Virtual methods for the OOZIP user hooks. You can override these virtual methods instead of using the SetXXX functions.

Syntax

```
function ZipFileList.Append(var CDH : CentralDirHead; var CP : CommentPtr;  
    var FName : PathStr) : Boolean;
```

Purpose

Add a node to the list.

Description

This method adds a new file to the end of the list, returning True if successful. CDH is the header for the file, CP is a pointer to the file comment, if any, and FName is the name of the file. Append is intended primarily for internal use by UnZip.BuildZipFileList.

See Also

UnZip.BuildZipFileList

ZipNode.Init

Syntax

```
procedure BuildZipFileList(var ZFL : ZipFileList; var FML : FileMaskList);  
procedure UnZip.BuildZipFileList(var ZFL : ZipFileList; var FML : FileMaskList);
```

Purpose

Build a list of files to be unzipped.

Description

This routine builds a ZipFileList containing all the files in the current ZIP file that match one or more of the file masks in the specified FileMaskList.

APZIP's BuildZipFileList returns 0 in ArchiveStatus if successful, else an error code. If no matching files are found, it returns epWarning+ecNoMatchingFiles. In the case of OOZIP, errors can be detected by calling the GetLastError method.

Examples

See the examples for InitZipFileList and ZipFileList.Init.

See Also

DoneZipFileList

ZipFileList.Init

Archive.GetLastError

InitZipFileList

Syntax

```
procedure CompressZip(Mask : PathStr);  
procedure Zip.Compress(Mask : PathStr);
```

Purpose

Compress all files that match a file mask.

Description

This is a simplified version of CompressFileMaskList. You may find this format more convenient when all of the files to be compressed can be described by a single mask.

See CompressFileMaskList for more information.

CompressFileMaskList / Create

Syntax

```
procedure CompressFileMaskListZip(var FML : FileMaskList);  
procedure Zip.CompressFileMaskList(var FML : FileMaskList);
```

Purpose

Compress all files that match a file mask list.

Description

This routine compresses all files that match one or more of the file masks in the specified FileMaskList.

CompressFileMaskList first attempts to compress each file using the compression method set by the last call to SetCompressionMode. If the compressed file size is larger than the original file size, then the compressed image is discarded and the file is simply copied into the archive.

See CreateZipFile for a small example. See the EXZIP and EXZIPO programs (earlier in this section and on disk) for simple but complete examples of using CompressFileMaskList.

See Also

Compress

SetCompressionMode

Syntax

```
constructor Zip.Create(FName : PathStr);
```

Purpose

Create a new ZIP archive file.

Description

This procedure creates a new ZIP archive file. It creates and initializes the ZIP archive using the standard ZIP format used by PKZIP. Use Create in lieu of Init when instantiating a Zip object associated with a new archive file.

An extension of ZIP is assumed if none is specified.

This constructor fails with ecCannotCreate if an archive with the specified name already exists. If any DOS I/O errors are encountered, the function fails and ArchiveStatus contains the DOS error code.

Note that if you create a new archive but fail to add any files to it, a 0 length file is left on the disk. It's your responsibility to delete such files (see ZIP/ZIPO for an example of this situation).

Example

```
var  
  Z : Zip;  
  Result : Word;  
begin  
  Z.Create('MYZIP.ZIP');  
  if ArchiveStatus <> ecOk then begin  
    WriteLn('Failed to create archive, error: ', ArchiveStatus);  
    Halt;  
  end;  
  Z.Compress('*.EXE');  
  Result := Z.GetLastError;  
  if Result <> ecOk then  
    WriteLn('Failed due to error ', Result);  
  Z.Done;  
end;
```

Creates a new archive file named MYZIP.ZIP and adds all files with the extension EXE to it.

See Also

UnZip.Done

UnZip.Init

Syntax

```
procedure CreateZipFile(ZipName : PathStr);
```

Purpose

Create a new ZIP archive file.

Description

This procedure creates a new ZIP archive file. It creates and initializes the ZIP archive using the standard ZIP format used by PKZIP.

An extension of ZIP is assumed if none is specified.

This function fails with `ecCannotCreate` if an archive with the specified name already exists. If any DOS I/O errors are encountered, the procedure fails and `ArchiveStatus` contains the DOS error code.

If you create a new archive but fail to add any files to it, a 0 length file is left on the disk. It's your responsibility to delete such files (see ZIP/ZIPO for an example of this situation).

Example

```
CreateZipFile('MYZIP.ZIP');
if ArchiveStatus <> ecOk then begin
  WriteLn('Failed to create archive, error: ', ArchiveStatus);
  Halt;
end;
Compress('*.EXE');
if ArchiveStatus <> ecOk then
  WriteLn('Failed due to error ', ArchiveStatus mod 10000);
DoneZipFile;
```

Creates a new archive file named MYZIP.ZIP and adds all files with the extension EXE to it.

See Also

DoneZipFile

InitZipFile

Syntax

```
procedure ZipFileList.Delete(ZNP : ZipNodePtr);
```

Purpose

Delete the specified node from the ZipFileList.

Description

This procedure deletes the specified node from the list and releases the memory associated with it. Note that the data associated with the node is permanently lost.

Example

```
ZFL.Delete(ZFL.zfHead);
Deletes the first node in the list.
```

See Also

DeleteZipFileListNode

DeleteFileMaskList / DeleteFiles

Syntax

```
procedure DeleteFileMaskListZip(var FML : FileMaskList);  
procedure Zip.DeleteFileMaskList(var FML : FileMaskList);
```

Purpose

Remove all files that match any mask in the file mask list from an archive.

Description

This procedure removes files from an existing archive. The files to be removed are described by the specified FileMaskList.

Example

```
var  
  Z : Zip;  
  FML : FileMaskList;  
  Result : Word;  
begin  
  FML.Init;  
  if not FML.Append('*.PAS') or  
    not FML.Append('*.EXE') then  
    {handle error} ;  
  Z.Init('MYZIP.ZIP');  
  if ArchiveStatus <> ecOk then  
    {handle error} ;  
  Z.DeleteFileMaskList(FML);  
  Result := Z.GetLastError;  
  if Result <> ecOk then  
    {handle error} ;  
  Z.Done;  
end.
```

Removes all files with extensions of PAS or EXE from MYZIP.ZIP.

See Also

DeleteFiles

Syntax

```
procedure DeleteFilesZip(Mask : PathStr);  
procedure Zip.DeleteFiles(Mask : PathStr);
```

Purpose

Remove all files that match a file mask from an archive.

Description

This is a simplified version of DeleteFileMaskList. You may find this more convenient when all of the files to be deleted can be described by a single mask.

See DeleteFileMaskList for more information.

Syntax

procedure DeleteZipFileListNode(var ZFL : ZipFileList; ZNP : ZipNodePtr);

Purpose

Delete the specified node from the ZipFileList.

Description

This procedure deletes the specified node from the ZipFileList and releases the memory associated with it. Note that the data associated with the node is permanently lost.

Example

```
DeleteZipFileListNode(ZFL, ZFL.Head);
```

Deletes the first node in the list.

See Also

ZipFileList.Delete

Syntax

destructor ZipFileList.Done; virtual;

Purpose

Destroy a ZipFileList.

Description

This destructor destroys a ZipFileList, releasing all the memory allocated to the individual nodes in the list.

Example

See the example for Init.

See Also

DoneZipFileList

Init

Syntax

destructor ZipNode.Done; virtual;

Purpose

Destroy a node.

Description

This destructor deallocates the node's memory. It is intended primarily for internal use by ZipFileList.Done.

See Also

ZipFileList.Done

DoneZipFile / DoneZipFileList / Extract

Syntax

procedure DoneZipFile;

Purpose

Close the ZIP file.

Description

This routine closes the ZIP file opened by InitZipFile or CreateZipFile.

Example

See the example for InitZipFile.

See Also

CreateZipFile

InitZipFile

Syntax

procedure DoneZipFileList(var ZFL : ZipFileList);

Purpose

Destroy a ZipFileList.

Description

This routine destroys a ZipFileList created by previous calls to InitZipFileList and BuildZipFileList.

Example

See the example for InitZipFileList.

See Also

BuildZipFileList

InitZipFileList

Syntax

procedure ExtractZip(Mask : PathStr);
procedure UnZip.Extract(Mask : PathStr);

Purpose

Extract all files that match the specified file mask.

Description

This procedure extracts all files from the ZIP file that match the specified Mask. Mask can be a specific file, or it can be a file mask.

Example

```
var
  UZ : UnZip;
begin
  if not UZ.Init('FOO.ZIP') then
    {... handle error ...};
  UZ.Extract('*.');
  if UZ.GetLastError <> 0 then
    {... handle error ...};
  UZ.Done;
end.
```

This program extracts all files in FOO.ZIP.

Syntax

```
procedure ExtractFileMaskListZip(var FML : FileMaskList);  
procedure UnZip.ExtractFileMaskList(var FML : FileMaskList);
```

Purpose

Extract all files that match the file mask list.

Description

This routine extracts all files that match one or more of the file masks in the specified FileMaskList. The procedure in APZIP stores a status code in ArchiveStatus: 0 if successful, else an error code. In the case of OOZIP, errors can be detected by calling the GetLastError method.

Examples

See the examples for InitZipFile and UnZip.Init.

See Also

ExtractZipFileList	InitFileMaskList (APARCHIV)
Archive.GetLastError	

Syntax

```
procedure ExtractZipFileList(var ZFL : ZipFileList);  
procedure UnZip.ExtractZipFileList(var ZFL : ZipFileList);
```

Purpose

Extract all files in the specified ZipFileList.

Description

This function extracts all files in ZFL, previously created by calling BuildZipFileList.

Note that, in most cases, you should use ExtractFileMaskList instead of this routine. ExtractZipFileList is preferred only in cases where you need to use the list of files to be extracted for some other purpose (e.g., displaying a list of files to be extracted before they are actually extracted).

Examples

See the examples for InitZipFileList and ZipFileList.Init.

See Also

BuildZipFileList	Archive.GetLastError
Extract	InitZipFileList
ExtractFileMaskList	

Syntax

```
procedure FreshenArchiveZip;  
procedure Zip.FreshenArchive;
```

Purpose

Conditionally update files in the current archive.

Description

This procedure conditionally updates all files in the current ZIP archive file. It processes the existing archive file sequentially. For each file in the archive, it looks for the same file in the current directory or the directory specified by the stored path name. If it finds such a file, it calls the `OkToCompress` function and passes it the ZIP header of the compressed file and the name of the matching file. The `OkToCompress` function decides which version belongs in the archive. If a file is not found on disk, the compressed file is transferred directly to the new archive.

If the `OkToCompress` function returns `True`, a new compressed image is created from the file on disk. If the `OkToCompress` function returns `False`, the existing compressed image is kept.

If no `OkToCompress` function was specified, all files are transferred to the new archive. The supplied function `DefOkToCompress` compares the old and new file dates and returns `True` only if the new file date is later than the old file date.

Example

```
var  
  Z : Zip;  
begin  
  Z.Init('MYZIP');  
  Z.FreshenArchive;  
  Z.Done;  
end;
```

A simple program to freshen all of the files in MYZIP.ZIP.

Syntax

```
constructor UnZip.Init(FName : PathStr);
```

Purpose

Initialize the archive and open the input file.

Description

This constructor attempts to open the specified ZIP file. It fails if the file cannot be opened successfully or if it is not a valid ZIP file.

Example

```
var  
  FML : FileMaskList;  
  UZ : UnZip;  
begin  
  FML.Init;  
  if not UZ.Init('FOO.ZIP') then  
    {... handle error ...};  
  UZ.ExtractFileMaskList(FML);  
  if UZ.GetLastError <> 0 then  
    {... handle error ...};  
  UZ.Done;  
  FML.Done;  
end.
```

This simplistic program extracts all files in FOO.ZIP.

See Also

Extract

InitZipFile

Syntax

```
constructor Zip.Init(FName : PathStr);
```

Purpose

Create a Zip archive.

Description

This constructor initializes the archive and opens the input file FName. It does this by calling UnZip.Init. See UnZip.Init for more information.

Syntax

```
constructor ZipFileList.Init;
```

Purpose

Initialize a ZipFileList.

Description

This constructor initializes a ZipFileList in preparation for a call to UnZip.BuildZipFileList.

Example

```
var
  FML : FileMaskList;
  ZFL : ZipFileList;
  UZ : UnZip;
begin
  FML.Init;
  ZFL.Init;
  if not UZ.Init('FOO.ZIP') then
    {... handle error ...};
  UZ.BuildZipFileList(ZFL, FML);
  if UZ.GetLastError <> 0 then
    {... handle error ...};
  UZ.ExtractZipFileList(ZFL);
  if UZ.GetLastError <> 0 then
    {... handle error ...};
  UZ.Done;
  ZFL.Done;
  FML.Done;
end.
```

This program extracts all files in FOO.ZIP. Simpler versions of this program can be found in the examples for UnZip.Init and UnZip.Extract.

See Also

InitZipFileList

Syntax

```
constructor ZipNode.Init(var CDH : CentralDirHead; var CP : CommentPtr;
                        var FName : PathStr);
```

Purpose

Initialize a node.

Description

This constructor initializes the node by storing the CDH, CP, and FName parameters and setting the znNext pointer to nil. It is intended primarily for internal use by ZipFileList.Append.

See Also

ZipFileList.Append

InitZipFile

Syntax

procedure InitZipFile(ZipName : PathStr);

Purpose

Open a ZIP file.

Description

This function opens the specified ZIP file and verifies that it is a ZIP file. If it isn't, ArchiveStatus is set to eeNotAZipFile. InitZipFile should be called before performing any APZIP file operations.

Note that the mode in which the file is opened depends on the current setting of Turbo Pascal's FileMode variable. If you want to open the file in read-only mode, set FileMode to \$20 before calling Init.

Example

```
var
  FML : FileMaskList;
  I : Word;
begin
  InitFileMaskList(FML);
  InitZipFile('FOO.ZIP');
  {... handle errors ...}
  ExtractFileMaskList(FML);
  {... handle errors ...}
  DoneZipFile;
  DoneFileMaskList(FML);
end.
```

This simplistic program extracts all files in FOO.ZIP.

See Also

DoneZipFile

UnZip.Init

Syntax

```
procedure InitZipFileList(var ZFL : ZipFileList);
```

Purpose

Initialize a ZipFileList.

Description

This procedure initializes a ZipFileList in preparation for a call to BuildZipFileList.

Example

```
var
  FML : FileMaskList;
  ZFL : ZipFileList;
  I : Word;
begin
  InitFileMaskList(FML);
  InitZipFileList(ZFL);
  InitZipFile('FOO.ZIP');
  {... handle errors ...}
  BuildZipFileList(ZFL, FML);
  {... handle errors ...}
  ExtractZipFileList(ZFL);
  {... handle errors ...}
  DoneZipFile;
  DoneZipFileList(ZFL);
  DoneFileMaskList(FML);
end.
```

This program extracts all files in FOO.ZIP. A simpler version of this program is in the example for InitZipFile.

See Also

BuildZipFileList
DoneZipFileList

ExtractZipFileList

Syntax

```
procedure SetCompressionModeZip(Mode : CompressionMode);
procedure Zip.SetCompressionMode(Mode : CompressionMode);
```

Purpose

Force the type of compression used.

Description

The Zip compression routines are capable of two types of compression: shrinking and imploding. By default, the best method is automatically selected when the file is compressed. This procedure overrides that selection by forcing the compression routines to shrink or implode only. It can also be used to reset to the best method algorithm.

Example

```
SetCompressionMode(cmImplode);
Force the Zip routines to implode only.
```

SetCompressSuccessFunc

Syntax

```
procedure SetCompressSuccessFuncZip(CSF : CompressSuccessFunc);  
procedure Zip.SetCompressSuccessFunc(CSF : CompressSuccessFunc);
```

Purpose

Specify a function to call after each file is compressed.

Description

Use this procedure to specify a CompressSuccess function. A CompressSuccess function is called after each file is compressed, giving you a chance to update your status display.

If using APZIP, the CompressSuccess function must be of the form:

```
{F+}  
function CompressSuccess(var CDH : CentralDirhead; FName : PathStr;  
                        ErrCode : Word) : Boolean;  
  
begin  
    ...  
end;
```

This function should return True if you want the compression process to continue.

If using OOZIP, the form should be:

```
{F+}  
function CompressSuccess(ZNode : ZipNodePtr; ZP : ZipPtr) : Boolean;  
  
begin  
    ...  
end;
```

This declaration also includes a pointer (ZP) to the object that made the call. You can use this pointer to obtain the results of the compression (with ZP^.GetLastError). You can override the virtual method OkToCompress in lieu of calling SetOkToCompressFunc.

Both OOP and non-OOP declarations include a central directory header (in the OOP declaration, the CDH header is a field of the ZipNodePtr—znCDH). This is the header of the file that was just compressed. The compression method can be found by examining ZNode^.znCDH.Method.

If the compress failed, then ErrCode (or ZP^.GetLastError) contains an Async Professional error code describing the cause of the failure. Generally, you should abort the compression process to restore the original archive for all errors—do this by having your CompressSuccess function return False. It is possible, however, to have the compression process continue after some errors. For example, if the compressor can't find a file specified in the file mask, it calls the CompressSuccess function with an ErrCode of ecFileNotFound. You may wish to simply display a message describing the problem and return True to have the update process continue with the rest of the file mask list.

Even if your CompressSuccess function returns True after an error, the compressor may decide to abort and restore the original archive anyway. It does this for all Turbo Pascal errors between 100 and 163 since those errors indicate a fatal programming error or a hardware problem.

A default CompressSuccess function called DefCompressSuccessFunc is supplied. This function updates the display, noting that the compression succeeded, and displays a ratio of compressed size to original size. Like all other default hooks in APZIP/OOZIP, the end result is a display that looks very similar to that produced by the PKZIP utility. DefCompressSuccessFunc returns False if ErrCode is non-zero, forcing the compression process to abort and the original archive to be restored.

Example

```
SetCompressSuccessFunc(DefCompressSuccessFunc);
```

This example uses the built-in function, DefCompressSuccessFunc, which displays the ratio of compressed file size to original file size.

Syntax

```
procedure SetExtractSuccessFuncZip(ESF : ExtractSuccessFunc);
procedure UnZip.SetExtractSuccessFunc(ESF : ExtractSuccessFunc);
```

Purpose

Specify a function to call after a file is unarchived.

Description

This procedure specifies a function that will be called immediately after the extraction of a file, giving you an opportunity to report any errors that occurred during the extraction process.

If using APZIP, the ExtractSuccess function should be of the form:

```
{SF+}
function ExtractSuccess(var CDH : CentralDirHead;
                        FName : PathStr;
                        ErrCode : Word) : Boolean;

begin
  ...
end;
```

The function should return False if it wants to abort the extract operation, either because of the nature of the error (e.g., disk full) or because the user asked to abort it. ErrCode contains 0 if the extraction was successful, else an error code. CDH is the header for the file that was being extracted, and FName is the name of the output file that was being created.

If using OOZIP:

```
{SF+}
function ExtractSuccess(ZNP : ZipNodePtr; FName : PathStr;
                        UP : UnZipPtr) : Boolean;

begin
  ...
end;
```

The code for the error encountered, if any, can be obtained by calling UP^.GetLastError. The header for the file can be accessed as ZNP^.znCDH.

Note that, when using OOZIP, you can override the virtual method ExtractSuccess in lieu of calling SetExtractSuccessFunc.

Example

```
SetExtractSuccessFunc(DefExtractSuccessFunc);
```

This example uses the provided DefExtractSuccess function, which reports and treats as non-fatal (i.e., not fatal to the entire extraction operation) the following errors: ecUnknownMethod, ecFileEncrypted, ecCannotCreate, and ecBadFileCRC. All other errors are considered fatal to the extraction operation.

SetFileCommentFunc

Syntax

```
procedure SetFileCommentFuncZip(FCF : FileCommentFunc);  
procedure Zip.SetFileCommentFunc(FCF : FileCommentFunc);
```

Purpose

Specify a function to call for file comments.

Description

Use this function to specify a function that will be called to get a new file comment after each file is compressed, or for each file that matches the file mask list supplied to UpdateCommentsFileMaskList.

If using APZIP, the FileComment function must be of the form:

```
{ $F+ }  
function FileComment(var CDH : CentralDirHead;  
                     FName : PathStr;  
                     var CP : CommentPtr;  
                     var Len : Word) : Boolean;  
  
begin  
    ...  
end;
```

If using OOZIP, the form should be:

```
{ $F+ }  
function FileComment(ZNode : ZipNodePtr;  
                     var CP : CommentPtr;  
                     var Len : Word;  
                     ZP : ZipPtr) : Boolean;  
  
begin  
    ...  
end;
```

This declaration also includes a pointer (ZP) to the Zip object that called this function. You can override the virtual method SetFileComment in lieu of calling SetFileCommentFunc. The OOP declaration includes a pointer to the current ZipNode (ZNode), which you can use to reference the central directory header (ZNode^.znCDH) and the file name (ZNode^.znFName).

If a file comment already exists, CP points to the comment and Len contains its length; otherwise, CP is nil and Len is zero.

If you want to replace the existing comment, this function should return True and CP should point to the new comment and Len should contain its length. The new comment must be allocated on the heap and must not be disturbed until after the current update process completes. The compressor automatically disposes of CP after it writes the central directory of the updated ZIP file (i.e., before CompressFileMaskList or FreshenArchive return).

If you want to delete the existing file comment without replacing it, this function should return True and CP should be set to nil.

If you want to keep the existing file comment, this function should return False.

The only restriction on the size of a file comment is that a file's name, comment, and extra data must total less than 65535 bytes. Async Professional does not attempt to enforce this requirement. In practice, however, you'll find very few programs that use the extra data area of a ZIP file and most of your comments will likely be short strings. So, it's unlikely you'll ever come close to this maximum allowable size.

Example

```
SetFileCommentFunc(DefFileCommentFunc);  
DefFileCommentFunc displays the old comment and prompts for a new one.
```

See Also

UpdateCommentsFileMaskList

Syntax

```
procedure SetImplodeFactors(MatchLength, Reps : Integer);  
procedure Zip.SetImplodeFactors(MatchLength, Reps : Integer);
```

Purpose

Set the factors that affect speed and compression ratios.

Description

This procedure allows you to adjust two internal variables used during the implode operation.

The default values are (4,4). Larger values tend to improve compression but increase the time needed to do the compression. A setting of (48,16) results in about the same compression as PKZIP. Larger values can achieve slightly better ratios with a speed penalty. MatchLength can be set between 4 and 288. Increase this value if the data you are compressing has runs of matching bytes longer than the current setting. The Reps parameter can range in value from 4 to 960. Increase the Reps setting if the data you are compressing has widely spaced matches. Files produced with any valid value for MatchLength and Reps can be decompressed using PKUNZIP.

Example

```
SetImplodeFactors(48,16);
```

Sets the match length to 48 and the repetition count to 16. These settings will yield a compression ratio equivalent to that of PKZIP in most cases. Causes the implode routine to search for runs of matches of at least 48 bytes long and to try this up to 16 times before accepting the current longest match.

SetOkToCompressFunc

Syntax

```
procedure SetOkToCompressFuncZip(OKC : OkToCompressFunc);  
procedure Zip.SetOkToCompressFunc(OKC : OkToCompressFunc);
```

Purpose

Specify a function to call before compressing a file.

Description

This function specifies an OkToCompress function—a function that is called before a file is compressed to give you the chance to skip the compression.

If using APZIP, the OkToCompress function must be global and of the form:

```
{F+}  
function OkToCompress(NewFile, OldFile : PathStr;  
                      var CDH : CentralDirHead) : Boolean;  
  
begin  
    ...  
end;
```

The function should return True if NewFile should be compressed, False if NewFile should be skipped.

If using OOOZIP, the form should be:

```
{F+}  
function OkToCompress(NewFile, OldFile: PathStr;  
                      var CDH : CentralDirHead;  
                      ZP : ZipPtr) : Boolean;  
  
begin  
    ...  
end;
```

When using OOOZIP, you can override the virtual method OkToCompress in lieu of calling SetOkToCompressFunc.

NewFile is the complete path name of the file that is to be compressed. OldFile is the name of the existing file in the archive (if there is one). If OldFile is non-blank, CDH contains the central directory header associated with that file.

The OkToCompress function is your opportunity to approve the addition of a file to the archive or to approve the replacement of an existing file.

Your OkToCompress function should handle the following two modes of operation:

1) If OldFile is blank, NewFile describes a new file to be compressed. You need to decide whether you want to add it to the archive. For example, you may want to exclude all .BAK files from the archive no matter what file masks are specified in the list. Note that if OldFile is blank, CDH is *not* initialized. Your OkToCompress function should return True if NewFile is to be added to the archive and False if it should be skipped.

2) If OldFile is not blank, NewFile and OldFile are the same file and you need to decide whether to replace the file (compress NewFile into the new archive) or leave the file as-is (copy OldFile from the old archive to the new archive). CDH contains the central directory header from the archive. You can reference any field in the header to help you decide whether to replace the file. For example, you would probably want to replace the file if the date/time stamp of NewFile is later than the date/time stamp from the CDH header.

Your OkToCompress function should return True if the file is to be replaced in the archive and False if the old file should be retained. Note that there is no way to skip an existing file—it will either be copied or replaced. If you need to delete an existing file, make a separate call to DeleteFiles or DeleteFileMaskList.

Async Professional provides a default OkToCompress function called DefOkToCompressFunc. For the first case described above, this default function always returns True. For the second case, it compares the date/time stamps of the two files and returns True if NewFile's date/time stamp is later than OldFile's date/time stamp, and returns False otherwise.

If you supply your own OkToCompress function, it must be declared exactly as above and be both global and far (using {`$F+`} or the far keyword). If you do not supply an OkToCompress function, new files are always added to the archive and matching files are always replaced.

Example

```
SetOkToCompressFunc(DefOkToCompressFunc);
```

This example uses the built-in DefOkToCompressFunc, which always returns True for new files and also returns True for existing files if the file on disk has a later date/time stamp.

SetOkToWriteFunc

Syntax

```
procedure SetOkToWriteFuncZip(OKF : OkToWriteFunc);  
procedure UnZip.SetOkToWriteFunc(OKF : OkToWriteFunc);
```

Purpose

Specify a function to call before extracting a file.

Description

This procedure specifies an OkToWrite function that will be called just before a file is extracted, giving you an opportunity to decide whether or not it should actually be extracted. The OkToWrite function can also change the name of the file to be written.

If using APZIP, the OkToWrite function should be of the form:

```
{ $F+ }  
function OkToWrite(var CDH : CentralDirHead; var FName : PathStr) : Boolean;  
begin  
    ...  
end;
```

The function should return False if the file should not be extracted (e.g., because it already exists). FName is the name of the output file that will be created if the file is extracted. The CDH parameter gives you access to the date/time stamp for the file. See the OkToWrite function in ZIPX.PAS for an example of how to use this information.

If using OOZIP:

```
{ $F+ }  
function OkToWrite(ZNP : ZipNodePtr; var FName : PathStr;  
                  UP : UnZipPtr) : Boolean;  
begin  
    ...  
end;
```

The header for the file can be accessed as ZNP^.znCDH. See the OkToWrite function in ZIPXO.PAS for a good example.

Note that, when using OOZIP, you can override the virtual method OkToWrite in lieu of calling SetOkToWriteFunc.

Example

```
SetOkToWriteFunc(DefOkToWriteFunc);
```

This example uses the supplied DefOkToWrite function, which always returns True.

Syntax

```
procedure SetShowCommentsProcZip(SCP : ShowCommentsProc);  
procedure UnZip.SetShowCommentsProc(SZCP : ShowCommentsProc);
```

Purpose

Specify a function to display ZIP file name/comments.

Description

Use this procedure to specify a routine that will be called at the beginning of any compression or extraction process, giving you an opportunity to display the name of the ZIP file and any comments concerning the ZIP file as a whole. The call is made after it has been determined that the file is a valid ZIP file, and before the scan for matching files begins.

If using APZIP, the ShowComments procedure should be of the form:

```
{ $F+ }  
procedure ShowComments(FName : PathStr; CP : CommentPtr;  
                       CLen : Word);  
  
begin  
    ...  
end;
```

FName is the name of the ZIP file used to build the ZipFileList. If CLen is not 0, CP is a pointer to a file comment (an array of char with a size of CLen).

If using OOZIP:

```
{ $F+ }  
procedure ShowComments(CP : CommentPtr; CLen : Word; UP : UnZipPtr);  
begin  
    ...  
end;
```

The name of the file can be obtained by calling UP^.GetFileName.

Note that, when using OOZIP, you can override the virtual method ShowComments in lieu of calling SetShowCommentsProc.

Example

```
SetShowCommentsProc(DefShowCommentsProc);
```

This example uses the provided DefShowCommentsProc, which simply writes 'Updating' (for compression) or 'Searching' (for extraction), followed by the filename passed to it (FName), and the file comment, if any.

See Also

BuildZipFileList

SetShowMethodProc

Syntax

```
procedure SetShowMethodProcZip(SMP : ShowMethodProc);  
procedure UnZip.SetShowMethodProc(SMP : ShowMethodProc);
```

Purpose

Specify a procedure to display the file name and the compression method.

Description

Use this procedure to specify a routine that will be called just before a file is compressed or extracted, giving you an opportunity to display its name and the compression method used.

If using APZIP, the ShowMethod routine should be of the form:

```
{SF+}  
procedure ShowMethod(Method : Byte; FName : PathStr);  
begin  
  ...  
end;
```

Method will be in the range cmcStored to cmcImploded. FName is the name of the output file that is created; it may or may not be a complete pathname.

If using OOZIP:

```
{SF+}  
procedure ShowMethod(ZNP : ZipNodePtr; FName : PathStr; UP : UnZipPtr);  
begin  
  ...  
end;
```

The compression method in this case is ZNP^.znCDH.Method.

Note that, when using OOZIP, you can override the virtual method ShowMethod in lieu of calling SetShowMethodProc.

Example

```
SetShowMethodProc(DefShowMethodProc);
```

This example uses the provided DefShowMethodProc, which writes 'Extracting' (Method = cmcStored), 'UnShrinking' (Method = cmcShrunk), 'Expanding' (Method = cmcReduced1..cmcReduced4), 'Exploding' (Method = cmcImploded), or 'Deleting' followed by the filename passed to it (FName).

During compression, DefShowMethodProc does *not* display the compression method. It is displayed in the OkToCompress function instead. When DefShowMethodProc is called, the compression method is not yet known and APZIP/OOZIP have not yet determined whether the file is new to the archive or just being updated. The DefOkToCompressFunc displays either "Shrinking" or "Imploding."

Syntax

```
procedure SetShowProgressFuncZip(SPF : ShowProgressFunc);  
procedure UnZip.SetShowProgressFunc(SPF : ShowProgressFunc);
```

Purpose

Specify a function to call to show progress.

Description

This procedure specifies a ShowProgress function that will be called periodically during the compression or extraction of a file, giving you an opportunity to indicate to the user that progress is being made. During compression this function is called after 4K bytes of the source file are processed. During extraction this function is called each time that the 8K output buffer is flushed to disk.

If using APZIP, then ShowProgress function should be of the form:

```
{SF+}  
function ShowProgress(BytesWritten, TotalBytes : LongInt) : Boolean;  
begin  
    ...  
end;
```

If using OOZIP:

```
{SF+}  
function ShowProgress(UP : UnZipPtr;  
                      BytesWritten, TotalBytes : LongInt) : Boolean;  
begin  
    ...  
end;
```

The function should return False if it wants to abort the compress or extract operation (e.g., because the user has pressed ^Break or ^C). During compression, BytesWritten is the number of original input bytes compressed so far and TotalBytes is the original size of the file. During extraction, BytesWritten is the number of bytes written so far and TotalBytes is the (uncompressed) size of the file being extracted. In the final call to ShowProgress, BytesWritten may be slightly smaller than TotalBytes (never reaching 100% complete). The subsequent call to CompressSuccess or ExtractSuccess is your indication that all bytes were processed (100% complete).

Note that, when using OOZIP, you can override the virtual method ShowProgress in lieu of calling SetShowProgressFunc.

Example

```
SetShowProgressFunc(DefShowProgressFunc);
```

This example uses the provided DefShowProgressFunc. During compression it displays the percentage of the file that has been compressed. During extraction it simply writes a '.' to the screen.

Syntax

```
procedure ZipNode.SetTag(On : Boolean);
```

Purpose

Tag/untag this node.

Description

This method tags or untags the file associated with this node. If the file is not tagged, it cannot be extracted from the ZIP file. All files are tagged by default by ZipNode.Init.

SetZipComment / UpdateCommentsFileMaskList

Syntax

```
procedure SetZipComment(var Comment; Len : Word);  
procedure Zip.SetZipComment(var Comment; Len : Word);
```

Purpose

Add or update a ZIP comment.

Description

This procedure adds or updates a ZIP comment. A ZIP comment is a single comment associated with a ZIP archive file.

Comment is an untyped parameter that contains the new ZIP comment. Len contains the length of this comment. The comment can be any block of data you desire. In most cases, however, you will probably pass a Pascal string. Unlike file comments, you are responsible for disposing of this variable if it is a heap variable.

To delete an existing ZIP comment, pass a Len of zero.

Example

```
var  
  MyComment : String;  
begin  
  MyComment := 'This is a zip comment';  
  SetZipComment(MyComment[1], Length(MyComment));  
  ...  
  Sets the ZIP comment to 'This is a zip comment'.
```

Syntax

```
procedure UpdateCommentsFileMaskList(var FML : FileMaskList);  
procedure Zip.UpdateCommentsFileMaskList(var FML : FileMaskList);
```

Purpose

Call the FileComment hook for all files in the archive that match FML.

Description

This procedure updates the file comments for specified files (or all files) within a ZIP archive.

In most cases you'll probably manage file comments when files are added to the archive (via the FileComments hook). In some cases, however, you may need to update the comments of files already in an archive. This is the purpose of UpdateCommentsFileMaskList.

This procedure examines every file in the archive, calling your FileComment hook for every file that matches a mask in FML. Your FileComment function can request that the comment be updated, deleted or left alone (see SetFileCommentFunc for more information).

This routine returns without doing anything if you fail to supply a FileComment function.

Example

```
FML.Init;  
if not FML.Append('*.*) then  
  {handle error} ;  
Z.Init('MYZIP');  
Z.SetFileCommentFunc(DefFileCommentFunc);  
Z.UpdateCommentsFileMaskList(FML);  
...
```

Calls the default FileComment function for every file in the archive MYZIP.ZIP.

See Also

SetFileCommentFunc

E. Demonstration Programs

Twelve small example programs are provided to illustrate the use of the modules described in this chapter.

- LZH and LZHO compress files into an LZH archive.
- LZHV and LZHVO view the directory of an LZH archive.
- LZHX and LZHXO extract files from an LZH archive.
- ZIP and ZIPO compress files into a ZIP archive.
- ZIPV and ZIPVO view the directory of a ZIP archive.
- ZIPX and ZIPXO extract files from a ZIP archive.

The first program listed in each pair above is written using the functions in APARCHIV, APLZH, and APZIP. The second program (the one that ends in 'O') uses the objects in OOARCHIV, OOLZH, and OOZIP. Although the implementation of the two programs is radically different, the two programs are identical from the user's point of view.

1. LZH Archive File Compress: LZH/LZHO

These two programs allow you to compress files into new or existing LZH archives. The procedural version (LZH) and the object-oriented version (LZHO) are used in exactly the same way. Each expects a command line of the following form:

```
LZH [Options] Filename[.LZH] files...
```

Filename is the name of an existing or new LZH archive to which files will be added. A file extension of LZH is assumed if none is specified. Note that the filename *cannot* include wildcard characters—these programs operate on one archive at a time.

The "files..." parameter allows you to specify the files to add to the archive. These parameters can be specific file names or file name masks and can include complete path name descriptions when necessary. For example, a files parameter of "READ.ME C:\AUTOEXEC.BAT *.PAS" adds the files READ.ME, C:\AUTOEXEC.BAT, and all files with a PAS extension to the archive. Note that you must enter at least one file name or file mask for all operations except freshening (F).

The following Options can be specified:

/A	add files to the archive
/D	delete files from the archive
/F	freshen (conditionally update) all files in the archive
/S	strip the path from the stored filename
/C	request confirmations
/?	display help screen

Note that '-' can be used instead of '/' when specifying command line options.

The /A parameter adds new files to the archive and updates existing files if the file in the archive is older than the file on disk. No confirmation is requested unless /C is specified.

The /D parameter deletes all files that match the specified file masks from the archive. No confirmation is requested—all matching files are deleted. Note that the /C parameter does not cause a confirmation in this case.

The /F parameter updates all files in the archive if the file in the archive is older than the corresponding file on disk. No confirmation is requested unless /C is specified.

The /S parameter tells the compressor to strip the path information from the file name before storing the file name in the header. Note that this may result in duplicate and ambiguous files in the archive.

The /C parameter requests user confirmation for each file to be added (or updated) to the archive. The confirmation options are <Y>, compress the file, <N>, skip the file (keeping the old file if there is one), <A>, stop asking for confirmation and assume that all subsequent files should be added to the archive, and <Q>, abort the update and restore the original archive.

Here are some examples of LZH/LZHO command lines:

```
LZH /a save *.*
```

Compresses all files in the current directory to the archive SAVE.LZH.

```
LZHO /d save *.bak
```

Removes all .BAK files from the archive SAVE.LZH.

```
LZH /f save
```

Freshens all files in the archive SAVE.LZH. Note that no file masks are required (if any are specified, they will be ignored).

```
LZHO /a mylzh c:\*.bat *.bat
```

Adds all batch files from the root directory on drive C: and from the current directory to the archive MYLZH.LZH. The batch files from drive C: are stored with their fully qualified names (e.g., C:\AUTOEXEC.BAT).

```
LZHO /a /s mylzh c:\*.bat *.bat
```

Adds all batch files from the root directory on drive C: and from the current directory to the archive MYLZH.LZH. The batch files from drive C: are stored *without* their pathnames. Note that this will create duplicate and ambiguous files in the archive if any batch files in the two directories have the same names.

2. LZH Archive Directory View: LZHV/LZHVO

As indicated above, these two utility programs allow you to view the directory of an LZH archive. The procedural version (LZHV) and the object-oriented version (LZHVO) are used in exactly the same way. Each expects a command line of the following form:

```
LZHV [Options] Filename[.LZH] [files...]
```

Filename is the name of the LZH archive whose directory is to be viewed. A file extension of LZH is assumed if none is specified. You can view the directory of a self-extracting archive created with LHARC's S option by specifying the full name of the executable file (e.g., INSTALL.COM or INSTALL.EXE). The Filename parameter can contain DOS wildcard characters ('?' or '*'). For example, to view the directories of all LZH archives in the current directory, you would enter LZHV * or LZHV *.LZH.

The optional [files...] parameter allows you to specify which files in the archive's directory you are interested in. For example, if you enter *.PAS, only files with a PAS extension are displayed. You can enter multiple filenames, any or all of which can contain DOS wildcard characters. If no file masks are specified, *.* is assumed (i.e., all files in the archive).

Currently, there are only two available options:

```
/T          technical (detailed) listing
/?          display help screen
```

The help screen displayed when you specify /? (or ?, or when you enter no parameters at all) simply reminds you of the program's command line syntax and the meanings of the optional parameters.

If the /T parameter is specified, the program uses the longer, "technical" format for displaying directory information. For example, here is a partial directory listing for an LZH file in the technical format:

Searching OPDIALOG.LZH

```
      Filename: DIALOG1.PAS
      Attributes: a--w
      Date and time: Oct 26, 1993  16:02:18
      Compression method: -lh1-
      Compressed size: 2121
      Uncompressed size: 5965
      16-bit CRC value: 13F7
```

```
      Filename: DIALOG2.PAS
      Attributes: a--w
      Date and time: Oct 26, 1993  16:02:18
      Compression method: -lh1-
      Compressed size: 2364
      Uncompressed size: 6606
      16-bit CRC value: 609C
```

```
      Filename: DIALOG3.PAS
      Attributes: a--w
      Date and time: Oct 26, 1993  16:02:18
      Compression method: -lh1-
      Compressed size: 4907
      Uncompressed size: 13591
      16-bit CRC value: D701
```

```
      Filename: DIALOG4.PAS
```

```

Attributes: a--w
Date and time: Oct 26,1993 16:02:18
Compression method: -lh1-
Compressed size: 2492
Uncompressed size: 6862
16-bit CRC value: DA93

```

The first line in each entry shows the name of the file. The second shows the file's attributes: 'a' means that the archive bit is set, 's' means the file is a system file, 'h' means it is a hidden file, 'r' means it is a read-only file, 'w' means that it can be written to. The third line shows the date and time when the file was created. The fourth line shows the compression method used: '-lh5-' means the file was compressed in LHA 2.x format, '-lh1-' means the file was compressed in LHARC format, '-lh0-' means that it was stored uncompressed. The fifth line shows the size of the compressed version of the file stored in the archive, and the sixth line shows its original size. And the last line shows the CRC value calculated for the file before it was compressed.

If the /T parameter is *not* specified, the program displays the directory information in a condensed format:

Searching OPDIALOG.LZH

Length	Method	Size	Ratio	Date	Time	CRC	Attr	Name
5965	-lh1-	2121	65%	10-26-93	16:02	13F7	a--w	DIALOG1.PAS
6606	-lh1-	2364	65%	10-26-93	16:02	609C	a--w	DIALOG2.PAS
13591	-lh1-	4907	64%	10-26-93	16:02	D701	a--w	DIALOG3.PAS
6862	-lh1-	2492	64%	10-26-93	16:02	DA93	a--w	DIALOG4.PAS
33024		11884	65%					4

As you can see, the non-technical format displays the same basic information as the technical format, as well as some additional information, such as the compression ratio. (A ratio of 65% means that the compressed version of the file is 65% smaller than the uncompressed file.)

Here are some examples of LZHV/LZHVO command lines:

```
LZHV mylzh *.pas
```

Display information about all files in MYLZHLZH with an extension of PAS.

```
LZHVO mylzh *.exe *.com *.bat
```

Display information about all files in MYLZHLZH with an extension of EXE, COM, or BAT.

```
LZHV /t mylzh | MORE
```

Display a listing of all the files in MYLZHLZH using the technical format. Output is piped to the DOS MORE utility so that it can be viewed a page at a time.

```
LZHVO * *.exe
```

Display a listing of all the EXE files in all the LZH archives in the current directory (* is equivalent to *.LZH).

```
LZHV files.exe
```

Display a listing of all files in FILES.EXE, a self-extracting LZH archive.

3. LZH Archive File Extract: LZHX/LZHXO

These two programs allow you to extract compressed files from an LZH archive. The procedural version (LZHX) and the object-oriented version (LZHXO) are used in exactly the same way. Each expects a command line of the following form:

```
LZHX [Options] Filename[.LZH] [files...]
```

Filename is the name of the LZH archive from which files are to be extracted. A file extension of LZH is assumed if none is specified. You can extract files from a self-extracting archive created with LHARC's S option by specifying the full name of the executable file (e.g., INSTALL.COM or INSTALL.EXE). The Filename parameter can contain DOS wildcard characters ('?' or '*'). For example, to extract all files from all LZH archives in the current directory, you would enter LZHX * or LZHX *.LZH.

The optional [files...] parameter allows you to specify which files in the archive's directory you want to extract. For example, if you enter *.PAS, only files with a PAS extension are extracted. You can enter multiple filenames, any or all of which can contain DOS wildcard characters. If no file masks are specified, *.* is assumed (i.e., all files in the archive).

The following Options can also be specified:

/D	create directories stored in LZH file
/N	extract only newer files
/O	overwrite existing files
/P path	path for output files
/?	display help screen

The /D parameter is used when extracting files whose names contain full or relative pathnames. It tells LZHX/LZHXO to create the specified directory if it doesn't already exist and to store the file in that directory. If the /D parameter is *not* specified, then the pathname is ignored. For example, suppose that the current directory is C:\TEMP, that the command is

```
LZHX /d mylzh
```

and that MYLZH.LZH contains a compressed file under the name DOCS\README.DOC. LZHX will write the file to C:\TEMP\DOCS\README.DOC, creating the C:\TEMP\DOCS directory if necessary. If the /D parameter were not specified, the file would be written to the current directory, C:\TEMP.

The /N parameter tells LZHX/LZHXO to extract files with the same name as an existing file *only* if the file in the archive has a more recent date/time stamp than the existing file.

The /O parameter tells LZHX/LZHXO to extract all matching files, whether or not they already exist. If the /O parameter is *not* specified, the program asks for confirmation before overwriting an existing file:

```
Warning: README.DOC exists. Overwrite it? (Y/N/A/Q) [N]
```

To tell the program to overwrite the file, press <Y>. To tell it not to overwrite the file, press <N> or <Enter>. Pressing <A> tells it to overwrite the file and to stop asking for confirmation. Pressing <Q>, <CtrlC>, or <CtrlBreak> aborts the program (Quit).

Note that the /O parameter does nothing if /N is also specified, since /N tells the program to overwrite existing files without asking for confirmation if the file in the archive is newer.

The /P parameter allows you to specify the drive:\directory where files are to be written when they are extracted. For example,

```
LZHX /p c:\temp mylzh
```

extracts all files from MYLZH.LZH and writes them to C:\TEMP.

The help screen displayed when you specify /? (or ?, or when you enter no parameters at all) simply reminds you of the program's command line syntax and the meanings of the optional parameters.

Here are some examples of LZHX/LZHXO command lines:

```
LZHX mylzh *.pas
```

Extract all files in MYLZH.LZH with an extension of PAS.

```
LZHXO mylzh *.exe *.com *.bat
```

Extract all files in MYLZH.LZH with an extension of EXE, COM, or BAT.

```
LZHX /o /p c:\temp mylzh
```

Extract all files in MYLZH.LZH, overwriting existing files without asking for confirmation. Extracted files will be written to C:\TEMP.

```
LZHXO /n * *.exe
```

Extract all the EXE files in all the LZH archives in the current directory (* is equivalent to *.LZH). If a given file already exists, it will be overwritten only if the archived file is newer than the existing file.

```
LZHX /o /d files.exe
```

Extract all files from FILES.EXE, a self-extracting LZH archive, without asking for confirmation before overwriting existing files. If the archive contains files with full or relative pathnames, the pathnames will be respected.

4. ZIP Archive File Compress: ZIP/ZIPO

These two programs allow you to compress files into new or existing ZIP archives. The procedural version (ZIP) and the object-oriented version (ZIPO) are used in exactly the same way. Each expects a command line of the following form:

ZIP [Options] Filename[.ZIP] files...

Filename is the name of an existing or new ZIP archive to which you wish to add files. A file extension of ZIP is assumed if none is specified. Note that the filename *cannot* include wildcard characters—these programs operate on one archive at a time.

The "files..." parameter allows you to specify what files you want to add to the archive. These parameters can be specific file names or file name masks and can include complete path name descriptions when necessary. For example, a files parameter of "READ.ME C:\AUTOEXEC.BAT *.PAS" adds the files READ.ME, C:\AUTOEXEC.BAT, and all files with a PAS extension to the archive. Note that you must enter at least one file name or file mask for all operations except freshening (/F).

The following Options can be specified:

/A	add files to the archive
/D	delete files from the archive
/F	freshen (conditionally update) all files in the archive
/K	prompt for a file comment for every existing file in ZIP file
/S	strip the path from the stored filename
/C	request confirmations
/Z	prompt for a ZIP comment
/P	prompt for file comments while compressing new files
/L	store drive letter with file
/M#	force compression mode
/?	display help screen

Note that '-' can be used instead of '/' when specifying command line options.

The /A parameter adds new files to the archive and updates existing files if the file in the archive is older than the file on disk. No confirmation is requested unless /C is specified.

The /D parameter deletes all files that match the specified file masks from the archive. No confirmation is requested—all matching files are deleted. Note that the /C parameter does not cause a confirmation in this case.

The /F parameter updates all files in the archive if the file in the archive is older than the corresponding file on disk. No confirmation is requested unless /C is specified.

The /K parameter makes a pass through the specified archive, prompting for a file comment for each file. If a file already has a comment associated with it, that comment will be displayed first. To keep the existing comment, press <Enter>. To delete the existing comment, press <Space> then <Enter>. To add a new comment or replace the old comment, type in the desired comment and press <Enter>.

The /S parameter tells the compressor to strip the path information from the file name before storing the file name in the header. Note that this may result in duplicate and ambiguous files in the archive.

The /C parameter requests user confirmation for each file to be added (or updated) to the archive. The confirmation options are <Y>, compress the file, <N>, skip the file (keeping the old file if there

is one), <A>, stop asking for confirmation and assume that all subsequent files should be added to the archive, and <Q>, abort the update and restore the original archive.

The /Z parameter prompts for a new ZIP comment. If the ZIP file already has a comment, then that comment is displayed first. To keep the existing comment, just press <Enter>. To delete the existing comment, press <Space> and <Enter>. To supply a new ZIP comment, type in the new comment and press <Enter>. The /Z option shouldn't be used in combination with any other option. If you do, the last option on the command line takes precedence.

The /P option prompts for file comments while adding or freshening files in the archive. For each new file that is compressed into the archive, you will be prompted for a file comment. Press <Enter> if you don't want a comment for that file.

The /L option indicates that the drive letter should be stored with the path. The default behavior of ZIP/ZIPO is not to store the drive letter.

The /M# option allows you to force a specific compression mode. Specify 1 for Store, 2 for Shrink, and 3 for Implode.

By default, ZIP and ZIPO use SetImplodeFactors(4,4) which results in an archive file slightly (1 to 5%) larger than the one PkZip would produce. Attempting to match or beat PKZIP's compression ratio by using larger values causes the algorithm to slow drastically.

Here are some examples of ZIP/ZIPO command lines:

```
ZIP /a save *.*
```

Compresses all files in the current directory to the archive SAVE.ZIP.

```
ZIPO /d save *.bak
```

Removes all .BAK files from the archive SAVE.ZIP.

```
ZIP /f save
```

Freshens all files in the archive SAVE.ZIP. Note that no file masks are required (if any are specified, they will be ignored).

```
ZIPO /a /l myzip c:\*.bat *.bat
```

Adds all batch files from the root directory on drive C: and from the current directory to the archive MYZIP.ZIP. The batch files from drive C: are stored with their fully qualified names (e.g., C:\AUTOEXEC.BAT).

```
ZIPO /a /s myzip c:\*.bat *.bat
```

Adds all batch files from the root directory on drive C: and from the current directory to the archive MYZIP.ZIP. The batch files from drive C: are stored *without* their pathnames. Note that this will create duplicate and ambiguous files in the archive if any batch files in the two directories have the same name.

```
ZIPO /z myzip
```

Displays the current ZIP comment and prompts for a new comment.

```
ZIP /k myzip
```

For every file in myzip, displays the current file comment and prompts for a new comment.

5. ZIP Archive Directory View: ZIPV/ZIPVO

Much like LZHV and LZHVO, these programs allow you to view the directory of a ZIP archive. The procedural version (ZIPV) and the object-oriented version (ZIPVO) are used in exactly the same way. Each expects a command line of the following form:

```
ZIPV [Options] Filename[.ZIP] [files...]
```

Filename is the name of the ZIP archive whose directory is to be viewed. A file extension of ZIP is assumed if none is specified. You can view the directory of a self-extracting archive created with ZIP2EXE by specifying the full name of the executable file (e.g., INSTALL.EXE). The Filename parameter can contain DOS wildcard characters ('?' or '*'). For example, to view the directories of all ZIP archives in the current directory, you would enter ZIPV * or ZIPV *.ZIP.

The optional [files...] parameter allows you to specify which files in the archive's directory you are interested in. For example, if you enter *.PAS, only files with a PAS extension are displayed. You can enter multiple filenames, any or all of which can contain DOS wildcard characters. If no file masks are specified, *.* is assumed (i.e., all files in the archive).

Currently, there are only two available options:

```
/T      technical (detailed) listing
/?      display help screen
```

The help screen displayed when you specify /? (or ?, or when you enter no parameters at all) simply reminds you of the program's command line syntax and the meanings of the optional parameters.

If the /T parameter is specified, the program uses the longer, "technical" format for displaying directory information. For example, here is a partial directory listing for an ZIP file in the technical format:

Searching OPDIALOG.ZIP - This is a zip file comment

```
      Filename: DIALOG1.PAS
      File type: text
      Attributes: --w
      Date and time: Oct 26,1993  16:02:18
      Compression method: Implode
      Compressed size: 2169
      Uncompressed size: 5965
      32-bit CRC value: 5D22ABB4
      Created by:   PKZIP 1.1
      Needed to extract: PKUNZIP 1.0
```

```
      Filename: DIALOG2.PAS
      File type: text
      Attributes: --w
      Date and time: Oct 26,1993  16:02:18
      Compression method: Implode
      Compressed size: 2399
      Uncompressed size: 6606
      32-bit CRC value: 4F333399
      Created by:   PKZIP 1.1
      Needed to extract: PKUNZIP 1.0
```

```
      Filename: DIALOG3.PAS
      File type: text
      Attributes: --w
      Date and time: Oct 26,1993  16:02:18
```

```

Compression method: Implode
  Compressed size: 4778
Uncompressed size: 13591
  32-bit CRC value: C4B7CD86
    Created by: PKZIP 1.1
  Needed to extract: PKUNZIP 1.0

  Filename: DIALOG4.PAS
  File type: text
  Attributes: --w
    Comment: This is a file comment
  Date and time: Oct 26,1993 16:02:18
Compression method: Implode
  Compressed size: 2517
Uncompressed size: 6862
  32-bit CRC value: 5D634497
    Created by: PKZIP 1.1
  Needed to extract: PKUNZIP 1.0

```

The first line in each entry shows the name of the file. The second shows the type of the file, according to PKZIP (binary or text). The third shows the file's attributes: 's' means the file is a system file, 'h' means it is a hidden file, 'r' means it is a read-only file, and 'w' means that it can be written to. The optional fourth line shows the comment associated with the file, if any. The fifth line shows the date and time when the file was created. The sixth line shows the compression method used: Stored, Shrunk, Reduced, or Imploded. The seventh line shows the size of the compressed version of the file stored in the archive, and the eighth line shows its original size. The ninth line shows the 32-bit CRC value calculated for the file before it was compressed. The tenth line shows the version of PKZIP used to compress the file, and the final line shows the version of PKUNZIP needed to extract it from the archive.

If the /T parameter is *not* specified, the program displays the directory information in a condensed format:

Searching OPDIALOG.ZIP - This is a zip file comment

Length	Method	Size	Ratio	Date	Time	CRC	Attr	Name
5965	Implode	2169	64%	10-26-93	16:02	5D22ABB4	--w	DIALOG1.PAS
6606	Implode	2399	64%	10-26-93	16:02	4F333399	--w	DIALOG2.PAS
13591	Implode	4778	65%	10-26-93	16:02	C4B7CD86	--w	DIALOG3.PAS
6862	Implode	2517	64%	10-26-93	16:02	5D634497	--w	DIALOG4.PAS
33024		11863	65%					4

As you can see, the non-technical format displays most of the same basic information as the technical format, as well as some additional information, such as the compression ratio. (A ratio of 65% means that the compressed version of the file is 65% smaller than the uncompressed file.)

Here are some examples of ZIPV/ZIPVO command lines:

```
ZIPV myzip *.pas
```

Display information about all files in MYZIP.ZIP with an extension of PAS.

```
ZIPVO myzip *.exe *.com *.bat
```

Display information about all files in MYZIP.ZIP with an extension of EXE, COM, or BAT.

ZIPV /t myzip | MORE

Display a listing of all the files in MYZIP.ZIP using the technical format. Output is piped to the DOS MORE utility so that it may be viewed a page at a time.

ZIPVO * *.exe

Display a listing of all the EXE files in all the ZIP archives in the current directory (* is equivalent to *.ZIP).

ZIPV files.exe

Display a listing of all files in FILES.EXE, a self-extracting ZIP archive.

6. ZIP Archive File Extract: ZIPX/ZIPXO

Like LZHX and LZHXO, these two programs allow you to extract compressed files from a ZIP archive. The procedural version (ZIPX) and the object-oriented version (ZIPXO) are used in exactly the same way. Each expects a command line of the following form:

```
ZIPX [Options] Filename[.ZIP] [files...]
```

Filename is the name of the ZIP archive from which files are to be extracted. A file extension of ZIP is assumed if none is specified. You can extract files from a self-extracting archive created with ZIP2EXE by specifying the full name of the executable file (e.g., INSTALL.EXE). The Filename parameter can contain DOS wildcard characters ('?' or '*'). For example, to extract all files from all ZIP archives in the current directory, you would enter ZIPX * or ZIPX *.ZIP.

The optional [files...] parameter allows you to specify which files in the archive's directory you want to extract. For example, if you enter *.PAS, only files with a PAS extension are extracted. You can enter multiple filenames, any or all of which can contain DOS wildcard characters. If no file masks are specified, *.* is assumed (i.e., all files in the archive).

The following Options can also be specified:

/D	create directories stored in ZIP file
/N	extract only newer files
/O	overwrite existing files
/P path	path for output files
/?	display help screen

The /D parameter is used when extracting files whose names contain full or relative pathnames. It tells ZIPX/ZIPXO to create the specified directory if it doesn't already exist and to store the file in that directory. If the /D parameter is *not* specified, then the pathname is ignored. For example, suppose that the current directory is C:\TEMP, that the command is

```
ZIPX /d myzip
```

and that MYZIP.ZIP contains a compressed file under the name DOCS\README.DOC. ZIPX will write the file to C:\TEMP\DOCS\README.DOC, creating the C:\TEMP\DOCS directory if necessary. If the /D parameter were not specified, the file would be written to the current directory, C:\TEMP.

The /N parameter tells ZIPX/ZIPXO to extract files with the same name as an existing file *only* if the file in the archive has a more recent date/time stamp than the existing file.

The /O parameter tells ZIPX/ZIPXO to extract all matching files, whether or not they already exist. If the /O parameter is *not* specified, the program asks for confirmation before overwriting an existing file:

```
Warning: README.DOC exists. Overwrite it? (Y/N/A/Q) [N]
```

To tell the program to overwrite the file, press <Y>. To tell it not to overwrite the file, press <N> or <Enter>. Pressing <A> tells it to overwrite the file and to stop asking for confirmation. Pressing <Q>, <CtrlC>, or <CtrlBreak> aborts the program (Quit).

Note that the /O parameter does nothing if /N is also specified, since /N tells the program to overwrite existing files without asking for confirmation if the file in the archive is newer.

The /P parameter allows you to specify the drive:\directory where files are to be written when they are extracted. For example,

ZIPX /p c:\temp myzip

extracts all files from MYZIP.ZIP and writes them to C:\TEMP.

The help screen displayed when you specify /? (or ?, or when you enter no parameters at all) simply reminds you of the program's command line syntax and the meanings of the optional parameters.

Here are some examples of ZIPX/ZIPXO command lines:

ZIPX myzip *.pas

Extract all files in MYZIP.ZIP with an extension of PAS.

ZIPXO myzip *.exe *.com *.bat

Extract all files in MYZIP.ZIP with an extension of EXE, COM, or BAT.

ZIPX /o /p c:\temp myzip

Extract all files in MYZIP.ZIP, overwriting existing files without asking for confirmation. Extracted files will be written to C:\TEMP.

ZIPXO /n * *.exe

Extract all the EXE files in all the ZIP archives in the current directory (* is equivalent to *.ZIP). If a given file already exists, it will be overwritten only if the archived file is newer than the existing file.

ZIPX /o /d files.exe

Extract all files from FILES.EXE, a self-extracting ZIP archive, without asking for confirmation before overwriting existing files. If the archive contains files with full or relative pathnames, the pathnames will be respected.

12. Appendices

A. Glossary of Communications Terms

This glossary contains a combination of industry accepted definitions and, where noted, definitions that are unique to Async Professional.

abstract layer

The lowest, most primitive layer in Async Professional. This layer is represented by the unit APPORT, which contains the abstract data and procedure definitions used by subsequent layers.

add-on layer

The highest layer of functionality in Async Professional. Units in this layer typically provide complex functions built using the lower layers. Examples include the modem and protocol units.

ANSI

American National Standards Institute. In Async Professional, references to ANSI usually refer to the ANSI standard for terminal control as exemplified by the DOS ANSI.SYS device driver.

asynchronous serial communication

Serially transmitted data in which each character is surrounded by start and stop bits. That is, each character can be extracted from the data stream without making assumptions, based on time, about when characters start and stop.

AT commands

An industry-standard set of commands for controlling modems introduced with the Hayes SmartModem.

baud rate

A measure of modulation rate, not communication speed. Technically, baud rate means the number of signal changes per second. At the UART, baud rate is generally equal to BPS (bits per second), since each signal change represents one bit. When using modems, however, baud rate is generally different than BPS, since the modulation schemes used by modems typically encode more than one bit per signal change. That's why modem speeds are typically rated in terms of BPS.

Bell 103

The AT&T modem standard for asynchronous communication at speeds up to 300 BPS.

Bell 212A

The AT&T modem standard for asynchronous communication at speeds up to 1200 bps on dial-up telephone lines.

BPS

Bits per second, a measure of raw communications speed, which quantifies how fast the bits within a character are being transmitted or received. It is not a measure of overall throughput, but rather a measure of the speed with which a single character can be processed.

break

A signal that can be transmitted or received over serial communication links. A break is not a character, but rather a condition in which the serial line is held in the "0" state for a least one character-time.

CAS

Communicating Applications Specification. A specification created by Intel and DCA (Digital Communications Associates, Inc.) providing a high-level interface for faxmodem control. This is the interface used by the Intel SatisFAXtion line of faxmodems.

CCITT

Comite Consultatif International de Telegraphique et Telephonique (International Telegraph and Telephone Consultative Committee). A European communications standards committee.

character-time

This term is used to mean the amount of time between the start bit and stop bit of a serial byte (inclusive). This is the smallest period of time between successive received or transmitted characters (of course, the elapsed time between characters can be longer than one character-time).

checksum

A byte, or bytes, appended to the end of a block of data that is used to check the integrity of that block. A checksum is the sum of all the bytes in the block.

Class 1, Class 2, Class 2.0 or Class I, Class II

Names for the TIA/EIA specifications for computer control of faxmodems. Class 1 is the older specification and places more work on software. Class 2 is more recent and places more work on the faxmodem. Note that Class 2.0 is not the same as Class 2. Class 2 refers to the non-ratified specification that many modem vendors used to produce the current generation of faxmodems. Class 2.0 refers to the ratified version of the specification.

com port

We use this term to refer to an asynchronous serial communications port.

core routines

A term to describe those few low-level Async Professional routines that every port must support. All higher-level routines are built from these low-level routines.

CRC

A byte, or bytes, appended to the end of a block of data that is used to check the integrity of the data. CRC is short for cyclical redundancy check, a data checking algorithm that provides a much higher level of protection than a simple checksum.

CTS

Clear to send. This is a modem control signal that is raised by the modem when it is ready to accept characters. The modem may lower this line when it cannot accept any more characters (usually, this means that its receive buffer is nearly full). This behavior is called hardware handshaking or hardware flow control.

data bits

The bits in a serial stream of data that hold data as opposed to control information. The number of data bits is one of the line parameters needed to describe a serial port configuration. The acceptable values are 5 through 8.

data compression (modems)

Refers to the ability of some modems to compress data before passing it to the remote modem. There are two standards that describe data compression methods, MNP and V.42bis.

DCD

Data carrier detect. A signal provided by a modem to indicate that it is currently connected to a remote modem.

DCE

Data communications equipment. Generally, this refers to a modem.

device layer

The second lowest layer of Async Professional, immediately above the abstract layer. It provides the physical connection between the software and the hardware.

DSR

Data set ready. This is a modem control input signal to a UART that tells the UART that the remote device (usually a modem) is active and ready to transmit data.

DTE

Data terminal equipment. Generally, this refers to a terminal or a PC emulating a terminal.

DTR

Data terminal ready. This is a modem control signal raised by a UART to notify the remote (usually a modem) that it is active and ready to transmit.

error correction (modems)

Refers to the ability of some modems to check the integrity of data received from a remote modem. There are two standards that describe error correction protocols, MNP and V.42.

fax

Common term for facsimile. Fax can refer to a device that sends or receives facsimile documents, the document itself, or the process of sending or receiving a document.

faxmodem

A modem with the ability, when used with appropriate software, to send and receive facsimile documents.

FIFO mode

A mode of operation for 16550 UARTs that takes advantage of the UART's first-in-first-out buffers.

floating view

A mode of operation in which a TerminalWindow object automatically adjusts the position of the viewport to try to keep incoming characters visible.

flow control

A facility that allows either side of a serial communication link to request a temporary pause in data transfer. Typically, such pauses are required when data is being transferred faster than the receiver can process it. Hardware flow control is implemented via changes in the CTS and RTS signals. Software flow control is implemented via the exchange of Xon and Xoff characters.

full duplex

1. A mode of communication in which the receiving computer automatically echoes all data it receives back to the transmitter. 2. A communications link that can pass data both directions (receive and transmit) at the same time.

Group 1,2,3

Standards for facsimile document transfer. The vast majority of faxmodems support only Group 3, the most modern standard governing document conversion and transmission between fax devices.

half duplex

1. A mode of communication in which the receiving computer does not echo any data back to the transmitter. 2. A communications link that can pass data in only one direction at a time.

handshaking

Refers to the initial transfers of data between two processes. Usually this term is used to describe the start of a protocol file transfer or the exchange of data that occurs when two modems first connect.

HDLC

High-level Data Link Control. A synchronous data link control protocol specification published by the International Standards Organization (ISO). Faxmodems use HDLC format to exchange information.

interface layer

The layer of Async Professional that contains the majority of the applications programming interface (API). This layer is implemented by OOCOM (OOP interface) and APCOM (non-OOP interface).

IRQ

One of the lines on the PC or PS/2 bus that is used to request a hardware interrupt. Any device that needs to interrupt the CPU (such as a UART) does so via an IRQ line.

kernel

In Async Professional this term is used to refer to the portions of APUART (or other device-layer units) that handle serial communications interrupts and directly access the UART registers. The kernel is also responsible for the automatic hardware and software flow control.

LAP M

An error-correction protocol included with the most recent CCITT communications standard V.42.

line error

Refers to one of the following errors: UART overrun, parity error, or framing error. Such errors are due either to interference picked up by the physical connection (cable, phone line, etc.) or to a mismatch in line parameters between the two ends of a serial link.

managed modem link

Refers to any modem protocol that manages the modem link (as opposed to passively moving data through the modem) and provides features such as error control and data compression.

MNP

Microcom Networking Protocol. A communications protocol designed by Microcom, Inc. and placed in the public domain. MNP defines several service levels that provide error control and data compression facilities between two modems. MNP will be of interest to you only if you are using modems that support it. In such a case, your modem manual will provide more information about the details of MNP.

modem

A device that facilitates serial communication over phone lines. The term is derived from the phrase MOdulation/DEModulation device.

network shared-modem pool

A collection of modems in a network that are available to any PC in the network. In a typical situation, several modems are attached to one PC (a "modem server") and other PCs on the network use a network protocol to access these modems.

parity

A bit that is used to check the integrity of a byte. The parity bit is set by the transmitter and checked by the receiver. If present, the parity bit is set so that the sum of the bits in the character is always odd or always even. The parity bit can also be set to a constant value (always on or always off).

pixel

A picture element. A picture is described by a collection of picture elements, bits that are either on or off. If a bit is on, that dot in the picture is black; if a bit is off, that dot in the picture is white.

port

In Async Professional, this short-cut term is sometimes used to refer to an asynchronous communications device in your PC (i.e., com port).

protocol

Generally, an agreed upon set of rules that both sides of a communications link follow. This term crops up in two places in Async Professional: file transfer protocols and modem protocols. A file transfer protocol is a set of rules that two computers use to transfer one or more files. A modem protocol describes the modulation technique as well as the error control and data compression rules.

raster line

A horizontal line of pixels, usually one line of a graphical image.

remote

In Async Professional this term is used to describe what's attached to your serial port. Since it may be another PC, a different kind of computer, a modem, an instrument, or another device, we often just say "remote" or "remote device."

RI

Ring indicator. A signal provided by the modem to indicate that a call is coming in (i.e., the phone is ringing).

RS-232

An EIA (Electronic Industries Association) standard that provides a physical description (voltages, connectors, pin names, and purposes) of a serial asynchronous communications link. This is the standard used by the IBM PC's Asynchronous Communications Adapter (and compatibles). The original intent of RS-232 was to describe the link between a computer and a modem. However, many devices other than modems (printers, plotters, laboratory instruments, and so on) have adopted some of the conventions of RS-232.

RTS

Request to send. This is a modem control signal that the UART uses to tell the modem that the UART is ready to receive data.

S-register

A register in a Hayes-compatible modem that stores configuration information. Lower numbered S-registers are somewhat standardized, but higher numbered S-registers are generally used for different purposes by different modem manufacturers.

serial data

Refers to data transmitted over a single wire where bits are represented as either high or low signals over a specified period of time. This is in contrast to parallel data, where each bit is represented by its own "wire."

start bit

The bit in a serial stream that indicates a data byte follows. This value cannot be changed; UART communications always uses one start bit.

stop bits

The bits in a serial stream that indicate all data bits were sent. One or two stop bits may be used. The number of stop bits is one of the line parameters needed to describe a serial link.

streaming protocol

A file transfer protocol that doesn't require an acknowledgement for each block. Such protocols are usually much faster than non-streaming protocols because the transmitter never pauses to wait for an acknowledgement.

terminal emulator

Software that interprets special sequences of characters as video control information (for setting colors, positioning the cursor, etc.) rather than data. This process is referred to as "emulation" because it emulates the behavior built into serial terminals (such as the DEC VT100 terminal).

terminal

A device (or software) that displays received data to a CRT and transmits keyboard characters to a host computer. A "dumb terminal" is one that does no local processing of the data it receives from the host. A "smart terminal" is capable of interpreting special "escape sequences," allowing the host to move the terminal's cursor, change the colors used to display text, etc.

TIA/EIA

Acronym for Telecommunications Industry Association / Electronic Industry Association—organizations that develop and maintain specifications for electronic and telecommunications devices.

UART

An acronym for Universal Asynchronous Receiver Transmitter. This is the device (usually one integrated circuit) that serializes and deserializes data between the CPU and the serial data line.

V.17

CCITT 7200, 9600, 12000, and 14400 bps faxmodem standard.

V.21

CCITT 300 bps faxmodem standard.

V.22

CCITT 1200 bps modem standard.

V.22bis

CCITT 2400 bps modem standard.

V.25bis

CCITT communications command set. Frequently implemented in addition to the AT command set.

V.27, V.27 ter

CCITT 2400 and 4800 bps faxmodem standard.

V.29

CCITT 7200 and 9600 bps faxmodem standard.

V.32

CCITT 9600 bps communications standard which describes a standard modem modulation technique. Any 9600 baud modem that complies with V.32 can connect to any other V.32 compliant modem. (This is an improvement from the early days of 9600 BPS communication when only modems from the same manufacturer could connect to each other.)

V.32bis

CCITT standard for data modem modulation rates up to 14400 bits per second.

V.42

CCITT error correcting protocol standard. Includes both MNP-4 and LAP-M error correction protocols.

V.42bis

CCITT 4:1 data compression protocol. This data compression scheme generally achieves a much higher degree of compression than is possible with MNP.

viewport

The portion of a TerminalWindow's virtual screen that is currently visible.

B. Error Codes

Async Professional uses a consistent set of rules throughout the library for defining and reporting error codes. A complete description of this system is provided in §4.I. The following is a brief reminder about how error codes are defined.

Error codes are stored in variables of type Word. The word is composed of an error type, which is a multiple of 10000; and an error number, which is the remainder. Given an error Code, you can determine its type and number with the following statements:

```
ErrorType := Code div 10000;  
ErrorNumber := Code mod 10000;
```

or, putting the Code back together again:

```
Code := (10000 * ErrorType) + ErrorNumber;
```

The error type should be used to determine the program's response to the error, i.e., whether to halt, prompt for a fix, warn and retry, etc. There are four error types and prefixes:

```
{Error types}  
etFatal    = 0;    {fatal errors}  
etNonFatal = 1;    {non-fatal I/O errors}  
etWarning  = 2;    {warnings, user input errors, etc.}  
etMessage  = 3;    {status information, simple messages}
```

```
{Error prefixes}  
epFatal    = etFatal    * 10000;  
epNonFatal = etNonFatal * 10000;  
epWarning  = etWarning  * 10000;  
epMessage  = etMessage  * 10000;
```

Error numbers provide more detail about an error, usually enough to determine why the error occurred. The lowest error numbers, ranging from 2 (file not found) to 162 (general failure), are the same as Turbo Pascal's I/O error codes. Other numbers are organized loosely in blocks of 1000, with each block indicating a different category of error:

0000-0999	DOS, Turbo Pascal, DOS critical errors
2000-2999	Capacity or environment errors
7000-7999	Warnings, user errors
8000-8999	Programmer errors
9000-9999	Messages and status codes

Note that the error numbers have been chosen to avoid conflicting with error numbers defined by Object Professional, which uses the same system.

The following is a complete list of all the error numbers. If you get one of these error numbers following a call to a procedure or method, see the documentation for that routine for more details. It is a good idea to refer to error numbers by name rather than by a specific numeric value (e.g., use 'ecTransmitFailed' rather than '2910'). The APMISC unit provides a function (StatusStr) that, given an Async Professional error code, returns a string that indicates the nature of the error (see §4.I).

DOS, Turbo Pascal, and DOS Critical Errors

```
{no error}  
ecOk                      = 0;    {Reset value for AsyncStatus}  
  
{DOS errors}  
ecFileNotFound            = 0002; {File not found}  
ecPathNotFound            = 0003; {Path not found}
```

ecTooManyFiles	= 0004;	{Too many open files}
ecAccessDenied	= 0005;	{File access denied}
ecInvalidHandle	= 0006;	{Invalid file handle}
ecOutOfMemory	= 0008;	{Insufficient memory}
ecInvalidDrive	= 0015;	{Invalid drive}
ecNoMoreFiles	= 0018;	{No more files}

{Turbo Pascal I/O errors}

ecDiskRead	= 0100;	{Attempt to read beyond end of file}
ecDiskFull	= 0101;	{Disk is full}
ecNotAssigned	= 0102;	{File not Assign-ed}
ecNotOpen	= 0103;	{File not open}
ecNotOpenInput	= 0104;	{File not open for input}
ecNotOpenOutput	= 0105;	{File not open for output}
ecInvalidFormat	= 0106;	{Invalid format for packed window}

{DOS critical errors}

ecWriteProtected	= 0150;	{Disk is write-protected}
ecUnknownUnit	= 0151;	{Unknown disk unit}
ecDriveNotReady	= 0152;	{Drive is not ready}
ecUnknownCommand	= 0153;	{Unknown command}
ecCrcError	= 0154;	{Data error}
ecBadStructLen	= 0155;	{Bad request structure length}
ecSeekError	= 0156;	{Seek error}
ecUnknownMedia	= 0157;	{Unknown media type}
ecSectorNotFound	= 0158;	{Disk sector not found}
ecOutOfPaper	= 0159;	{Printer is out of paper}
ecDeviceWrite	= 0160;	{Device write error}
ecDeviceRead	= 0161;	{Device read error}
ecHardwareFailure	= 0162;	{General failure}

Capacity or Environment Errors

{APUART port errors}

ecNoMorePorts	= 2900;	{Can't open port, no slots available}
ecOverrunError	= 2901;	{UART receiver overrun}
ecParityError	= 2902;	{UART receiver parity error}
ecFramingError	= 2903;	{UART receiver framing error}

{APINT14 port errors}

ecTransmitFailed	= 2910;	{Int14 transmit failed}
ecUartError	= 2911;	{Int14 receive failed}

{APCOM/OOCOM errors/status}

ecBlockIncomplete	= 2920;	{Block shorter than requested}
ecBufferIsFull	= 2921;	{No room for new char in buffer}
ecBufferIsEmpty	= 2922;	{No characters to get}
ecTimeout	= 2923;	{Timed out waiting for something}
ecStringIncomplete	= 2924;	{String shorter than requested}
ecStringOverrun	= 2925;	{String longer than 255}
ecUserAbort	= 2926;	{User aborted during "wait"}

{APMODEM/OOMODEM errors}

ecTableFull	= 2930;	{No room in table to add new entry}
ecNullCommand	= 2931;	{Modem - no command registered}

{Tracing/EventFile file errors}

ecEventFileError	= 2940;	{Failed to open or write to the event file}
ecTraceFileError	= 2941;	{Failed to open or write to the trace file}

{APFOSSIL port errors}

ecNoFossil	= 2950;	{No fossil driver installed}
------------	---------	------------------------------

{APDIGI14 port errors}

ecDigiFailure = 2960; {Generic Digiboard failure code}

Warnings and User Errors

{This category currently not used by Async Professional}

Programmer Errors

{APCOM/OOCOM port errors}

ecBadPortNumber = 8900; {Out-of-range port number}
ecOutOfRange = 8901; {General out-of-range error}
ecPortNotOpen = 8902; {Port not open}
ecInvalidBaudRate = 8903; {Bad baud rate for this device}
ecInvalidArgument = 8904; {General programming error}
ecNoDevice = 8905; {No device layer installed}
ecNotaUart = 8906; {Couldn't find a UART at this address}
ecInvalidParity = 8907; {Bad parity option for this device}
ecNotBuffered = 8910; {Operation only allowed on buffered ports}
ecNotSupported = 8911; {Function not supported by device layer}

Messages and Status Codes

{APFAX error codes}

ecFaxVoiceCall = 9800; {Call is VOICE}
ecFaxDataCall = 9801; {Call is DATA}
ecFaxBusy = 9802; {Called modem is busy}
ecFaxNoFontFile = 9803; {Could not find font file}
ecFaxNoCASManager = 9804; {CASMgr TSR not installed}
ecFaxInitError = 9805; {Unexpected response in init}
ecFaxTrainError = 9806; {Failed to train with remote modem}
ecFaxSessionError = 9807; {Error during session}
ecFaxNoConnect = 9808; {No connection after dial}
ecFaxPageError = 9809; {Failed to send page after retries}

{APABSPCL/OOABSPCL status codes}

ecInitFail = 9900; {Xmodem init failed}
ecInitCancel = 9901; {Xmodem init was canceled on request}
ecCancelRequested = 9902; {Cancel requested}
ecDuplicateBlock = 9903; {Duplicate block received}
ecSequenceError = 9904; {Wrong block number received}
ecDirNotFound = 9905; {Directory not found in protocol transmit}
ecNoMatchingFiles = 9906; {No matching files in protocol transmit}
ecLongPacket = 9907; {Long packet received during protocol}
ecEndFile = 9908; {End of transmitted file}
ecHandshakeInProgress = 9909; {Initial protocol handshake in progress}
ecFileRenamed = 9910; {Incoming file was renamed}
ecFileAlreadyExists = 9911; {Incoming file already exists}
ecInvalidFileSize = 9912; {Ymodem header has bad file size}
ecInvalidDateTime = 9913; {Ymodem header has bad date/time}
ecUnexpectedChar = 9914; {Unexpected char during protocol}
ecBlockCheckError = 9915; {Incorrect CRC or checksum received}
ecNoSearchMask = 9916; {No search mask specified for transmit}
ecNoFilename = 9917; {No filename specified in Xmodem download}
ecAsciiReceiveInProgress = 9918; {Ascii receive in progress}
ecFileRejected = 9919; {Receiver rejected file}
ecTooManyErrors = 9920; {Too many errors received during protocol}
ecBadFileList = 9921; {No end of list marker found in file list}

{APZMODEM/OOZMODEM status codes}

ecGotCrcE = 9925; {Zmodem - got CrcE DataSubpacket}
ecGotCrcW = 9926; {Zmodem - got CrcW DataSubpacket}
ecGotCrcQ = 9927; {Zmodem - got CrcQ DataSubpacket}

ecGotCrcG	= 9928;	{Zmodem - got CrcG DataSubpacket}
ecGarbage	= 9929;	{Zmodem - got garbage from remote}
ecSkipFile	= 9930;	{Zmodem - skip file}
ecBadPosition	= 9931;	{Zmodem - bad file position}
ecFileDoesntExist	= 9932;	{Zmodem - specified file doesn't exist}
ecCantWriteFile	= 9933;	{Zmodem - not allowed to overwrite file}
ecFailedToHandshake	= 9934;	{Zmodem - never got proper handshake}
ecNoFilesToReceive	= 9935;	{Zmodem - no files to receive}
ecBuffersTooSmall	= 9936;	{Zmodem - port buffers too small}
ecGotHeader	= 9937;	{Zmodem - got a complete header}
ecNoHeader	= 9938;	{Zmodem - (internal) no header yet}
 {APMODEM/OOMODEM status codes}		
ecUnknownModemResult	= 9940;	{Unexpected char in modem result string}
ecConnect	= 9941;	{Modem response - CONNECT}
ecRing	= 9942;	{Modem response - RING}
ecNoCarrier	= 9943;	{Modem response - NO CARRIER}
ecNoDialTone	= 9944;	{Modem response - NO DIALTONE}
ecBusy	= 9945;	{Modem response - BUSY}
ecNoAnswer	= 9947;	{Modem response - NO ANSWER}
ecRinging	= 9948;	{Modem response - RINGING}
ecVoice	= 9949;	{Modem response - VOICE}
ecError	= 9950;	{Modem response - ERROR}
 {APKERMIT/OOKERMIT status codes}		
ecGotData	= 9954;	{Kermit - got packet}
ecNoData	= 9955;	{Kermit - no data yet}
 {Archive status messages}		
ecUnknownMethod	= 9960;	{Unknown compression method}
ecFileEncrypted	= 9961;	{Cannot extract--file is encrypted}
ecBadFileCRC	= 9962;	{Bad CRC--file is probably corrupted}
ecCannotCreate	= 9963;	{Unable to create output file}
ecBadFileFormat	= 9964;	{Bad LZH file format}
ecNotAnLzhFile	= 9965;	{Not an LZH file}
ecNotAZipFile	= 9966;	{Not a ZIP file}
ecEmptyFileMaskList	= 9967;	{File mask list is empty}
 {FAX conversion error/warnings}		
ecFaxBadFormat	= 9970;	{Not a valid APRO fax file}
ecFontNotSupported	= 9971;	{Bad font format or unsupported feature}
ecHPFontCvt	= 9972;	{Converting font}
ecBadGraphicsFormat	= 9973;	{Bad TIFF/PCX format or unsupported feature}
ecBadEscapeSeq	= 9974;	{Bad escape sequence in PCL file}
 {CaptureTerminalWindow error codes}		
ecScrollBackTooBig	= 9980;	{Scroll back buffer > 64k}
 {APBPLUS/OOBPLUS status codes}		
ecResync	= 9985;	{Resyncing with host}
ecWaitACK	= 9986;	{Waiting for ACK}
ecDropout	= 9987;	{Dropout}
ecHostCan	= 9988;	{Host cancel}
ecTryResume	= 9989;	{Attempting resume}
ecHostResume	= 9990;	{Host resuming}
ecResumeOK	= 9991;	{Resumed OK}
ecResumeBad	= 9992;	{Failed to resume}
ecUnPacket	= 9993;	{Invalid packet type}

C. Debugging Communications Programs

This appendix discusses more fully some of the topics introduced in §1.H "Calling for Technical Support." Mostly, this is a list of tips and techniques for debugging communications programs. Some of these suggestions are very simple and you probably already use them. However, some of them are specific to communications programs and might cover issues you haven't had to deal with before.

First, always make sure that your hardware is set up correctly (check connections, cabling, switches, etc.). The best way to verify this is to start with a known, reliable communications program such as COMTEST. If COMTEST doesn't work, you know that there's something wrong with either the com port itself, the cable, the device you are connected to, or the line parameters you're using.

Using the Debugger

Assuming that you've gotten past this hurdle and into real programming issues, what do you do if your communications program isn't doing what it's supposed to? With non-communications programs, the general thing to do is to understand the circumstances that cause the problem and repeat them in the debugger (usually the Turbo Pascal integrated debugger or Turbo Debugger). Sometimes this is an iterative process which can take several steps to zero in on the problem area. However, most repeatable problems can be figured out with just a few minutes of debugger time.

Unfortunately, this standard process doesn't always work well with communications programs. The problem has to do with the debuggers themselves. When the debugger is running and on-screen, it has restored any vectors set by the program being debugged. This makes it extremely difficult, if not impossible, to step or trace through your program while serial I/O is occurring. While you are stepping through your program, the debugger is preventing the Async Professional ISR from seeing any incoming characters. To your program, it will be as if those characters never arrived. Since your program is probably evaluating this data and taking various actions based on what it received, it will probably consider the little data it does get to be garbage. (What happens after that depends on how well your application handles garbage data.)

The situation is even worse when transmitting data. Async Professional depends on an uninterrupted chain of transmit completion interrupts to transmit a buffer of data. If any of these interrupts are missed, Async Professional becomes hopelessly confused and stops transmitting data completely. That means that you can *never* step over or trace into a statement, such as 'PutString(S)', that transmits data.

These problems do not mean that you can't use a debugger at all; you just might have to make some changes in the way you debug your programs. Generally, you need to rely on breakpoints and you need to choose those breakpoints very carefully. The goal is never to have the debugger on the screen while serial I/O is taking place. Choose your breakpoints such that all serial I/O has already occurred (or, more likely, enough I/O has occurred to get you to the problem area in the program).

The problem with this method of debugging is that it usually fools the remote device into thinking that you're no longer connected (because once you've hit your breakpoint no more serial I/O can take place). You can't always resume the program at your end when you're ready to advance to some later breakpoint. Usually you'll have to restart both your program and your session with the remote before beginning another debugging run.

Despite this problem, a debugger is still a very useful tool for communications programmers. We relied on it heavily during the development of many parts of Async Professional by following the breakpoint philosophy outlined above.

Using the Async Professional Debugging Tools

Async Professional has several built-in features for aiding in the debugging process. The simplest, and probably most useful, is the Tracing facility. It provides a character-by-character audit report of all the data transmitted or received by your program. Tracing is particularly useful when your program advances to some point and then starts misbehaving. After a few minutes of study, a Trace of such a program run will generally lead you right to the problem area. See "Tracing" in §4.B for more information on this facility.

Async Professional provides another auditing tool called EventLogging, which works at a much lower level than Tracing. EventLogging provides an exact chronology (with microsecond timestamps) of all events that occur at the UART itself. It's handy for figuring out problems with hardware flow control and other control signal situations (e.g., "why isn't my program answering a ringing phone?"). See "EventLogging" in §4.C for more information on this facility.

Finally, Async Professional provides the capability for you to supply a User Error Procedure. This is a procedure that Async Professional calls whenever it detects any kind of error. All of your programs should include such an error procedure. With an error procedure in place, you are notified of all errors, even when you might not be specifically checking for them your program. See "User Error Procedure" in §4.B for more information.

Common Problems

Here's a brief discussion of some of the common problems that popped up during development and testing of Async Professional. They are organized in a question and answer format.

Nothing works, not even the supplied test programs. What's wrong?

Probably a hardware or cabling problem that you'll need to figure out before you go any further. Common problems are two or more UARTs using the same IRQ, another board (e.g., a mouse or network board) using a com port IRQ, or two or more UARTs using the same I/O address.

Why am I getting UART overruns?

A UART overrun occurs when a character is received at the serial port before the Async Professional ISR has a chance to process the previous character. That is, characters are coming too fast for Async Professional to handle them.

There is a finite limit to the speed at which a given machine can receive data. Reasonable limits for various processors are listed in "Baud Rate" in §2.B. It could be that you selected a baud rate that is too high for your machine.

A more likely cause, however, is that another process related directly or indirectly to your application is leaving interrupts off for too long. While interrupts are off, Async Professional isn't notified of incoming characters. If interrupts are left off for more than one character-time, it's very likely that you will lose characters due to UART overruns.

Interrupts can be left off in a variety of circumstances. Some EMS drivers leave interrupts off while remapping the EMS pages. This could become a problem if you are using a service that relies on EMS (such as the Turbo Pascal overlay manager or the Object Professional large array facility). Additionally, 386 memory managers like 386Max and QEMM increase the amount of time required to process a serial interrupt and they, too, occasionally leave interrupts off longer than your program may tolerate.

Generally, whenever you receive UART overruns, you should simplify both your environment (by removing EMS drivers and so on) and your application (by removing overlays) to try to isolate the cause of the problem.

Why do my protocol transfers seem slow?

This usually means that your status procedure is taking too much time. You shouldn't try to do any lengthy calculations, disk I/O, or any other time consuming activities in your status procedure. Again, this can sometimes happen indirectly if the call to your status routine requires an overlaid unit to be read from disk or EMS.

You can test this hypothesis fairly quickly by trying a test run without your status procedure or with a very simple status procedure instead.

Why am I getting parity and framing errors?

Either you're operating with a different set of line parameters than the remote device, or your cable is picking up interference. Generally, the higher the baud rate you select, the more likely you are to suffer from electrical interference. If you suspect that your cable is picking up interference from other electrical sources, consider rerouting the cable run away from such sources.

Why isn't my program answering a ringing phone?

If you are depending on the DeltaRI bit (which is our recommended approach), but your program never sees that the phone is ringing, your cable probably doesn't provide a connection for the Ring Indicator line. You can check this by generating an event log and looking for modem status interrupts that show when the DeltaRI bit is set. If the phone is ringing but the event log never shows a set DeltaRI bit, then your cable doesn't provide that line.

My protocol transfer never gets started. What's wrong?

This could be due to any of several problems: mismatched line parameters, wrong protocol selected, couldn't find a file to transmit or others. Your best bet is to generate a Trace and see just how far the protocol did get. Also, try one of the demonstration programs in the same situation to see if it works. Generally, this should provide enough information to find and correct the problem.

After I run an Async Professional program, the next communication program I run locks up (or behaves strangely). What's wrong?

Most likely, your program enabled the FIFO buffers on the UART and neglected to disable them before it ended. Some older communications programs don't deal well with enabled FIFO buffers.

How can I use non-standard COM port addresses or IRQs?

To use a non-standard base address or IRQ requires both hardware and software changes. The hardware change usually involves changing jumpers or dip switch settings on your serial port board. The software change is that your program must call *SetUart* *before* opening the serial port. For example, change the jumpers on your serial port board so that Com3 uses IRQ5, then call *SetUart* like so:

```
SetUart(Com3, $3E8, 5, $D);
```

This call must be made before the port is opened (whether using OOP or procedural syntax). Note also that *SetUart* simply informs Async Professional about non-standard assignments that you made within your hardware. If your hardware doesn't support such non-standard assignments, or you don't make the assignments, then *SetUart* can't do anything for you.

Why doesn't my program work when it tries to open Com1 and Com3 at the same time (or if a mouse is on Com3 when I open Com1)?

Because Com1 and Com3 share an interrupt line, IRQ4, you cannot open both ports at the same time on ISA bus machines (which most PCs and PC clones are). The problem is not software—it is that the ISA hardware was not designed to share IRQ lines. If two devices try to control one IRQ line at the same time, the state of the IRQ line is unpredictable.

Async Professional does include logic in its interrupt service routine for sharing IRQ lines, but this only helps on EISA and MCA (PS/2) machines. It's activated by removing the semicolon before `ShareIrqOn` in `APUART.ASM` and rebuilding the library with `APRO.MAK` or `APROO.MAK`. If you enable `ShareIrqOn`, you must close ports in the reverse order that you open them so that the interrupt chaining is "undone" correctly. Failing to do so may leave an incorrect ISR address in the shared vector after your program ends (which may cause a fatal crash soon after).

If you don't have an EISA or ISA machine and you want to use `Com1` and `Com3` at the same time, then you'll need to change the IRQ line of one of those devices first. See the previous question for more information on changing IRQs.

This discussion also applies to opening `Com2` and `Com4` at the same time since those ports also share an interrupt line (IRQ3).

My Zmodem file transfer program generates lots of `ecBlockCheck` errors and `ecLongPacket` errors, but other protocols work fine. What's going on?

The answer in this case is almost always lack of hardware flow control. The problem shows up in Zmodem but not other protocols because Zmodem is a streaming protocol. Data is sent in a continuous stream without pauses for acknowledgements. Flow control is required to prevent the sender from overflowing the modem or the receiver. And remember, flow control must be enabled at four places: your software, your modem, the remote software, and the remote modem. See §5.C for information on flow control. Consult your modem manual for the hardware flow control enable command for your modem.

A second, though less likely, possibility is that the protocol is getting a lot of `ecOverrunErrors`. If you've taken our advice, your error procedure is ignoring these errors during protocols, so it might not be obvious that this is the problem. As a quick check, have your error procedure report, or simply count, `ecOverrunErrors`. The most likely cause of this error is running at a baud rate that is too high for your environment. It can also be caused by a TSR or device driver leaving interrupts off too long. This is more likely if you're running a protected mode application or running under a multitasker such as Windows, OS/2, or DesqView. See "Line Errors and Breaks" in §2.B for suggestions for increasing performance in these cases.

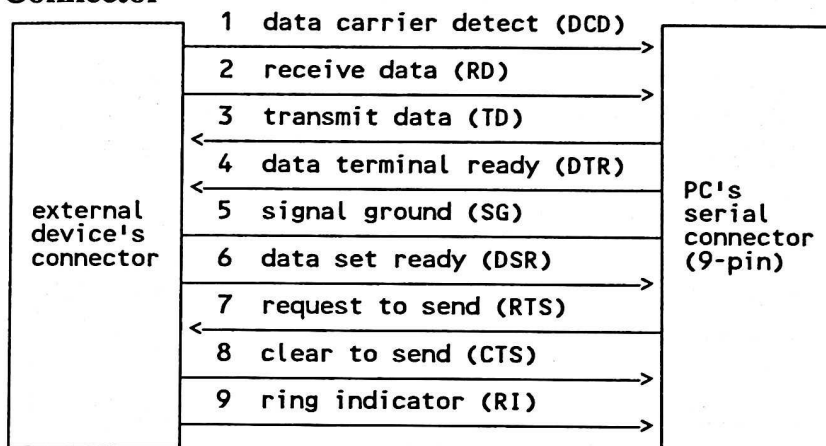
I'm writing a "door" program for a BBS, so I need to open a com port without changing its line parameters. How do I do this?

If you are using the object-oriented interface, open the port with `InitKeep` instead of `InitCustom`. If you are using the procedural interface, open the port with `InitPortKeep` instead of `InitPort`.

D. Serial Port Diagram

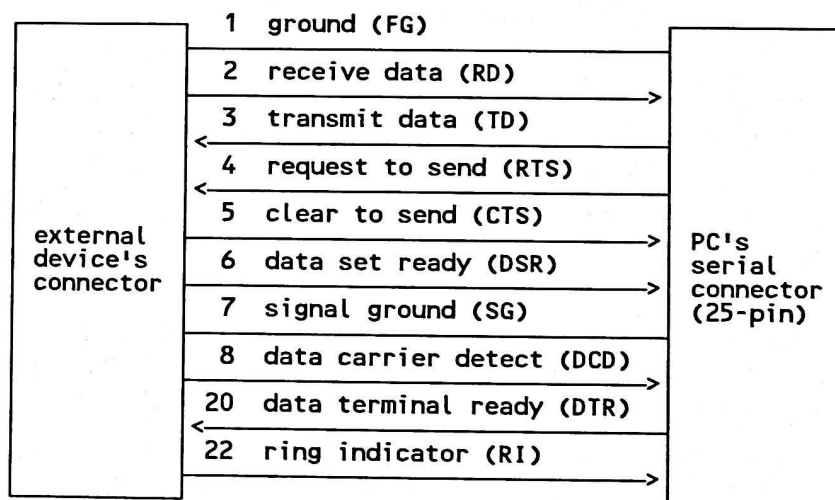
PC, AT, and PS/2 serial ports provide two different connection formats: 9-pin and 25-pin. Typically, the documentation supplied with your serial port boards will describe the type of connector required and a diagram of the pin designations. Just so that you'll have such information handy in one place, we provide standard connector diagrams below:

9-Pin Connector



25-Pin Connector

(shows only those pins used by standard UARTs)



E. What's New in Version 2.0

This section provides an overview of the new features in Async Professional version 2.0. It also describes a few changes that might impact your Async Professional 1.x programs.

Async Professional now provides fax capabilities. You can convert text, PCX, and TIFF documents to fax format. You can display and print received fax files, and even convert them to PCX. The fax send and receive units are modeled after the protocol units, with a similar set of features and user hooks. Async Professional supports Class 1, Class 2, and CAS send and receive faxmodems. See Chapter 10 for a complete description of the fax facilities.

Another new feature of Async Professional version 2.0 is DOS protected mode DLLs. Using Borland Pascal 7.0, you can compile Async Professional to a series of DLLs rather than directly linking the code. Naturally you are free to distribute the DLLs with your applications. See §2.F for more information on protected mode DLLs.

Two new device layer units were added. APFOSSIL provides support for FOSSIL device drivers. APDIGI14 provides support for DigiBoards using the DigiBoard XIDOS5.SYS device driver. The bonus DigiBoard unit from early versions of Async Professional is gone because APDIGI14 works with a much wider set of DigiBoard products. These changes first appeared in Async Professional 1.10 and 1.11. They are now also documented in the manual (see §4.D and §4.E).

Buffer information and software/hardware flow control functions were moved to the device layer from the interface layer.

The protocol layer of Async Professional now includes the CompuServe B+ protocol. The architecture of B+ closely matches that of other Async Professional protocols. The B+ bonus unit is gone, although the GIF support remains in the bonus archive GIFDEMO.LZH. See §7.H for a description of the B+ protocol.

All file transfer protocols now support the background model, which permits multitasking the protocols with another process in your program or writing true background protocol TSRs. See §7.C for more information on background file transfers.

The ZIP and LZH units now support compression in addition to decompression. ZIP 1.x implode compression is supported. On the LZH side, support was added for both compression and decompression of LHA 2.x files. The compression units are documented in Chapter 11.

If you are upgrading from version 1.x of Async Professional, you will notice a few changes. The biggest change is that Turbo Pascal 6.0 is now required. This is because many of the new routines take advantage of features that are available in Turbo Pascal 6.0 and later.

If you wrote a device layer using Async Professional 1.03 or earlier, you need to add some additional core functions in the area of buffer information and flow control. See 74.G for more information on the core routines.

A misnamed public identifier was changed: NoAbortProc was renamed to NoAbortFunc.

Identifier Index

When groups of related constants are documented, only the first constant in the group is listed in the Identifier Index. For example, `afAbortNoConnect` is in the index, but the other constants in that group (`afCASWaitTillDone`, `afExitOnError`, and `afCASSubmitUseControl`) are not.

A

- `AbortCurrentEvent` 596
- `AbortFunc` 106
- `AbortTracing` 111
- `AbstractFax` 548
- `AbstractFaxConverter` 511
- `AbstractModem` 255
- `AbstractPort` 187
- `AbstractProtocol` 319
- `AcceptFaxFunc` 547, 548
- `AcceptFileFunc` 319, 320
- `ActivateApDigi14` 139
- `ActivateApFossil` 136
- `ActivateApInt14` 141
- `ActivateApUart` 126
- `ActivatePort` 110, 143
- `ActivatePortProc` 106
- `ActiveComPort` 110
- `AddFaxEntry` 549
- `AddFileToList` 321
- `AddModemCode` 256
- `AddModemCommand` 257
- `AddModemResponse` 258
- `AddTraceEntry` 111
- `afAbortNoConnect` 546
- `afOptionsAreOn` 550
- `afOptionsOff` 550
- `afOptionsOn` 550
- `AnsiEmulator` 458
- `AnswerModem` 259
- `apIncludeDirectory` 316
- `apOptionsAreOn` 321
- `apOptionsOff` 321
- `apOptionsOn` 322
- `Append (FileMaskList)` 637
- `Append (Lzh)` 658
- `Append (Zip)` 697
- `AppendFileMask` 637
- `AproFontName` 509
- `apSuppressCtrlZ` 426
- `Archive` 636
- `ArchiveStatus` 168
- `arCreateDirs` 634
- `arOptionsAreOn` 637
- `arOptionsOff` 638
- `arOptionsOn` 638
- `arOutPath` 635

- `AsciiProtocol` 427
- `AssignPortDev` 444
- `AsyncErrorProc` 106
- `AsyncStatus` 168
- `AutoAnswerModem` 260
- `AutoDeviceInit` 12

B

- `BadArchiveOptions` 635
- `BadFaxCvtOptions` 509
- `BadFaxOptions` 547
- `BadPortOptions` 102
- `BadProtocolOptions` 317
- `BadTerminalWinOptions` 473
- `BadUnpackOptions` 509
- `Basic` 11
- `bcNone` 317
- `bcsNone` 317
- `BindFaxFont` 12
- `BlockFillChar` 317
- `BlockReady` 188
- `BPlusOverHead` 413
- `BPlusProtocol` 414
- `BPlusTurnDelay` 413
- `BPtr` 106
- `BufferFlush` 110, 143
- `BufferFlushProc` 106
- `BufferStatus` 110, 144
- `BufferStatusProc` 106
- `BuildLzhFileList` 658
- `BuildZipFileList` 697

C

- `C12AbstractFax` 562
- `C12ReceiveFax` 562
- `C12SendFax` 562
- `CalibrateDelay` 89
- `Capture` 487
- `CaptureIsOn` 487
- `CaptureTerminalWindow` 474
- `CASFax` 595
- `CASInstalled` 596
- `cclncomingChar` 473
- `cCR` 103
- `CentralDirHead` 691
- `CentralDirTail` 692
- `ChangeBaud` 188
- `ChangeBufferSizes` 189
- `ChangeDataBits` 189
- `ChangeParity` 190
- `ChangeStopBits` 190
- `CharArray` 106
- `CharReady` 110, 144
- `CharReadyFunc` 106
- `CharSet` 106
- `CheckCTS` 191
- `CheckDataReady` 191
- `CheckDCD` 192

- CheckDeltaCTS 192
- CheckDeltaDCD 193
- CheckDeltaDSR 193
- CheckDeltaRI 194
- CheckDSR 194
- CheckFifoError 195
- CheckForString 112
- CheckLineBreak 196
- CheckLineError 196
- CheckRI 197
- CheckTE 197
- CheckTHRE 197
- ClassifyUart 126
- ClearFaxEntries 551
- ClearFlag 162
- ClearScreenProc 451
- ClearTracing 113
- cLF 102
- CloseFile 596
- cmcStored (Lzh) 653
- cmcStored (Zip) 691
- CommandRecord 458
- ComNameString 113
- ComNameType 106
- Compress (Lzh) 659
- Compress (Zip) 697
- CompressFileMaskList (Lzh) 659
- CompressFileMaskList (Zip) 698
- CompressionMode (Lzh) 654
- CompressionMode (Zip) 693
- CompressSuccess (Lzh) 654, 655, 657
- CompressSuccess (Zip) 693, 694, 695
- ControlFileRecord 594
- ConvertFax (PcxFaxConverter) 513
- ConvertFax (TextFaxConverter) 513
- ConvertFax (TiffFaxConverter) 513
- ConvertFaxPcx 513
- ConvertFaxText 513
- ConvertFaxTiff 513
- CopyVirtToFile 476
- CountsPerMs 88
- CourierCodeSet 251
- CourierCommandSet 251
- CourierModem 255
- CourierResponseSet 251
- Crc (Lzh) 655
- Crc (Zip) 694
- Create (Lzh) 660
- Create (Zip) 698
- CreateLzhFile 661
- CreateOutputFile 638
- CreateZipFile 699
- CurrentAnsiPort 451
- cXoff 103
- cXon 103

D

- DataBitType 106

- DataBlockType 318
- DeactivatePort 110, 145
- DefArchiveOptions 635
- DefAsciiOptions 427
- DefaultLineOptions 103
- DefaultXoffChar 103
- DefaultXonChar 103
- DefBaseAddr 125
- DefBlockLen 426
- DefBlockWait 351
- DefCaptureBufferSize 473
- DefCaptureFileName 473
- DefComVector 125
- DefConnectAttempts 547
- DefDelayFactor 250
- DefDialTimeout 250
- DefDigi14Options 138
- DefEOLChar 426
- DefFaxCvtOptions 509
- DefFaxOptions 547
- DefFinishRetry 372
- DefFinishWait (Xmodem) 351
- DefFinishWait (Zmodem) 372
- DefFossilOptions 134
- DefHandshakeRetry 317
- DefHandshakeWait 317
- DefHibitPrefix 388
- DefInlt 547
- DefInterCharDelay 427
- DefInterLineDelay 427
- DefIrqNumber 125
- DefKermitOptions 388
- DefMaxBlockErrors 351
- DefMinRepeatCnt 388
- DefPortOptions 103
- DefProtocolOptions 317
- DefRcvTimeout 427
- DefReceiveTimeout 372
- DefScrollBackCols 473
- DefScrollBackRows 473
- DefSepChar 251
- DefSig 159
- DefStatusInterval 317
- DefTerminalWinOptions 473
- DefTimeout 251
- DefTransTimeout 318
- DefUnpackOptions 509
- Delay 89
- DelayTics 89
- Delete 661
- Delete (ZipFileList) 699
- DeleteAllFiles 597
- DeleteFile 597
- DeleteFileMaskList (Lzh) 662
- DeleteFileMaskList (Zip) 700
- DeleteFiles (Lzh) 662
- DeleteFiles (Zip) 700
- DeleteLzhFileListNode 663
- DeleteZipFileListNode 701

- DeltaCTSMask 104
- DialModem 261
- Digi14Port 187
- DisableStatusBuffer 198
- DisposeFileList 322
- Done (AbstractFax) 551
- Done (AbstractModem) 262
- Done (AbstractProtocol) 322
- Done (AnsiEmulator) 462
- Done (Archive) 639
- Done (AsciiProtocol) 429
- Done (BPlusProtocol) 416
- Done (C12AbstractFax) 563
- Done (C12ReceiveFax) 563
- Done (C12SendFax) 563
- Done (CaptureTerminalWindow) 487
- Done (CASFax) 598
- Done (FileMaskList) 639
- Done (FileMaskNode) 639
- Done (KermitProtocol) 391
- Done (LzhFileList) 663
- Done (LzhNode) 663
- Done (PcxFaxConverter) 514
- Done (TerminalEmulator) 459
- Done (TerminalWindow) 476
- Done (TextFaxConverter) 514
- Done (TiffFaxConverter) 514
- Done (UnpackFax) 523
- Done (YmodemProtocol) 364
- Done (ZipFileList) 701
- Done (ZipNode) 701
- Done (ZmodemProtocol) 375
- DoneAscii 429
- DoneBPlus 416
- DoneC12ReceiveFax 564
- DoneC12SendFax 564
- DoneCASFax 598
- DoneFileMaskList 639
- DoneKermit 391
- DoneLzhFile 664
- DoneLzhFileList 664
- DoneModem 262
- DonePcxConverter 514
- DonePort 110, 145
- DonePortProc 106
- DoneTextConverter 514
- DoneTiffConverter 514
- DoneUnpacker 523
- DoneUnpackToPcx 523
- DoneXmodem 353
- DoneYmodem 364
- DoneZipFile 702
- DoneZipFileList 702
- DoneZmodem 375
- DQDefault 414
- DrainingStatusInterval 372
- DrainOutBuffer 199
- DTRDropHold 251
- DTRMask 104

- DumpEvents 127
- DumpTrace 114
- DumpTraceHex 114

E

- ecOk 167
- ElapsedTime 90
- ElapsedTimeInMSecs 90
- ElapsedTimeInSecs 91
- EnableStatusBuffer 200
- eNone 457
- ErrorStates 254
- ESCIResponse 414
- EstimateTransferSecs 323
- EventLogging 13
- EventTimer 88
- ExecuteModemCommand 263
- ExpandFileMaskList 640
- ExternalDataBlock 594
- Extract (UnLzh) 665
- Extract (UnZip) 702
- ExtractFileMaskList (Lzh) 665
- ExtractFileMaskList (Zip) 703
- ExtractLzhFileList 666
- ExtractSuccess (Lzh) 654, 655, 657
- ExtractSuccess (Zip) 693, 694, 695
- ExtractZipFileList 703

F

- FastAbort 388
- FaxConverter 511
- FaxCvtStatusFunc 511
- FaxFileExt 159
- FaxHeaderRec 510
- FaxInProgress 200
- FaxLogProc 548
- FaxNameFunc 548
- FaxRec 548
- FaxReceive (C12) 565
- FaxReceive (CAS) 599
- FaxReceivePart (C12) 566
- FaxReceivePart (CAS) 600
- FaxStateType 547
- FaxStatusProc 548
- FaxTransmit (C12) 567
- FaxTransmit (CAS) 601
- FaxTransmitPart (C12) 568
- FaxTransmitPart (CAS) 602
- fcDoubleWidth 509
- fcOptionsAreOn 515
- fcOptionsOff 515
- fcOptionsOn 515
- ffHighRes 510
- FifoStatus 127
- FileBufferArray 318
- FileBufferSize 318
- FileComment 693, 694
- FileListType 318

FileMaskList 635, 636
 FileMaskNode 635, 636
 FileTransferRecord 594
 FindFirstCAS 603
 FindNextCAS 603
 FlagsSet 162
 FlowCtlProc 107
 FlowGetFunc 107
 FlowSetProc 107
 FlowState 107
 FlushInBuffer 201
 FlushOutBuffer 201
 ForceBufferLimits 202
 FossilPort 187
 FreeMemCheck 163
 FreshenArchive (Lzh) 666
 FreshenArchive (Zip) 704

G

GetAllStatus 604
 GetArchiveStatus 169
 GetAsyncStatus 169
 GetBaseAddr 202
 GetBlock 203
 GetBlockDirect 204
 GetBlockErrors 323
 GetBlockNum 324
 GetBlockSize 324
 GetBlockTimeout 204
 GetBytesRemaining 325
 GetBytesTransferred 325
 GetCaptureFileName 488
 GetChar 110, 146
 GetCharProc 107
 GetCharTimeout 205
 GetCheckType 326
 GetComName 205
 GetConnectSpeed 263
 GetCurrentBlockNum 326
 GetDelimLoc 206
 GetElapsedTics 327
 GetEventDate 605
 GetEventStatus 605
 GetEventTime 606
 GetExternalDataBlock 606
 GetFaxHeader 524
 GetFaxName 552
 GetFaxProgress 553
 GetFileName (Archives) 640
 GetFileName (Fax converter) 516
 GetFileName (Protocols) 327
 GetFileSize 328
 GetHangupResult 569
 GetHardwareStatus 607
 GetIncomingChar 477
 GetInitialFilePos 328
 GetLastCode 264
 GetLastError 640

GetLastErrorMode 264
 GetLastPageStatus 571
 GetLastText 265
 GetLine 110, 146
 GetLineControl 206
 GetLineError 207
 GetLineNumber 429
 GetLineProc 107
 GetLineStatus 207
 GetLPStatus 392
 GetMemCheck 163
 GetModem 110, 147
 GetModemClassSupport 572
 GetModemControl 208
 GetModemFeatures 573
 GetModemInfo 574
 GetModemProc 107
 GetModemRegister 265
 GetModemResponse 266
 GetModemStatus 208
 GetPageHeader 524
 GetPageInfo (C12) 575
 GetPageInfo (CAS) 607
 GetPathname 329
 GetPortDevPtr 445
 GetProtocol 330
 GetQueueStatus 608
 GetRemoteID 575
 GetSessionParams 576
 GetSetAutoReceive 608
 GetStatusInfo 516
 GetString 209
 GetStringTimeout 210
 GetSwcSize 392
 GetTerminalWinEmulator 477
 GetTerminalWinPort 478
 GetTotalErrors 330
 GmtHourOffset 318
 GotError 110, 147
 GotErrorProc 107

H

HandleResumeProc 414
 HandshakeWait 372
 HangupModem 267
 HayesCodeSet 251
 HayesCommandSet 251
 HayesModem 255
 HayesResponseSet 251
 hfUseDTR 104
 HWFlowDisable 210
 HWFlowEnable 211
 HWFlowGet 110, 148
 HWFlowSet 110, 148
 HWFlowState 212
 HWOnly 11

I

- InBuffFree 213
- InBuffUsed 213
- InhibitAutoLF 451
- InhibitModeChange 451
- Init (AbstractFax) 553
- Init (AnsiEmulator) 462
- Init (Archive) 641
- Init (AsciiProtocol) 430
- Init (BPlusProtocol) 416
- Init (C12AbstractFax) 577
- Init (C12ReceiveFax) 577
- Init (C12SendFax) 577
- Init (CaptureTerminalWindow) 488
- Init (CASFax) 609
- Init (FileMaskList) 641
- Init (FileMaskNode) 641
- Init (HayesModem) 267
- Init (KermitProtocol) 393
- Init (Lzh) 667
- Init (LzhFileList) 667
- Init (LzhNode) 667
- Init (PcxFaxConverter) 517
- Init (TerminalEmulator) 459
- Init (TerminalWindow) 478
- Init (TextFaxConverter) 517
- Init (TiffFaxConverter) 517
- Init (UnLzh) 668
- Init (UnpackFax) 525
- Init (UnZip) 704
- Init (XModemProtocol) 353
- Init (YModemProtocol) 365
- Init (Zip) 705
- Init (ZipFileList) 705
- Init (ZipNode) 705
- Init (ZmodemProtocol) 376
- InitAscii 430
- InitBPlus 417
- InitC12ReceiveFax 578
- InitC12SendFax 578
- InitCASFax 609
- InitCustom (AbstractProtocol) 331
- InitCustom (AsciiProtocol) 431
- InitCustom (BPlusProtocol) 417
- InitCustom (CaptureTerminalWindow) 489
- InitCustom (KermitProtocol) 393
- InitCustom (TerminalWindow) 479
- InitCustom (XModemProtocol) 354
- InitCustom (YModemProtocol) 365
- InitCustom (ZmodemProtocol) 376
- InitCustomAscii 431
- InitCustomBPlus 418
- InitCustomKermit 394
- InitCustomXmodem 355
- InitCustomYmodem 366
- InitCustomZmodem 377
- InitEventLogging 128

- InitFast 214
- InitFileMaskList 641
- InitKermit 394
- InitLzhFile 668
- InitLzhFileList 669
- InitModem 268
- InitModemCourier (APCOUR) 293
- InitModemForFaxReceive 579
- InitModemMcom (APMCOM) 294
- InitPcxConverter 517
- InitPort 110, 149
- InitPortFast 214
- InitPortKeep 110, 150
- InitPortKeepProc 107
- InitPortProc 107
- InitTextConverter 517
- InitTiffConverter 517
- InitTracing 115
- InitUnpacker 525
- InitUnpackToPcx 525
- InitXmodem 355
- InitYmodem 366
- InitZipFile 706
- InitZipFileList 707
- InitZmodem 377
- Int14Port 187
- IsPS2 117

K

- KermitOptionRec 389
- KermitOverhead 388
- KermitProtocol 389, 390
- KermitTurnDelay 389

L

- LargeComNameSet 12
- LineOptionRecord 107
- Load (AbstractModem) 269
- Load (AnsiEmulator) 463
- Load (AsciiProtocol) 432
- Load (BPlusProtocol) 419
- Load (CaptureTerminalWindow) 490
- Load (Digi14Port) 215
- Load (FossilPort) 215
- Load (Int14Port) 215
- Load (KermitProtocol) 395
- Load (TerminalEmulator) 459
- Load (TerminalWindow) 480
- Load (UartPort) 215
- Load (XModemProtocol) 356
- Load (YModemProtocol) 367
- Load (ZmodemProtocol) 378
- LoadFont 518
- LoadHPFont 519
- LocalHeader 692
- LogFileProc 319, 320
- LogFileType 318
- Lzh 656

LzhFileList 654, 656
LzhHeader 654
LzhNode 655, 656

M

MakeFileList 331
MaskBlinkOnInverse 451
Match (FileMaskList) 642
Match (FileMaskNode) 642
MatchFileMask 643
MaxActivePort 104
MaxBadBlocks 372
MaxCoverData 594
MaxDialStr 251
MaxSRegs 251
MaxStdSRegs 251
MaxWindowSlots 389
MaxWordLen 251
mcNone 252
mCodeResults 252
mEchoOff 252
MicrocomCodeSet 251
MicrocomCommandSet 251
MicrocomModem 255
MicrocomResponseSet 251
MinCaptureBufferSize 473
MinInBuff 104
ModemCodeTable 254
ModemCommandTable 254
ModemRec 255
ModemRegType 254
ModemResponseTable 254
MoveReceivedFile 610
mrOk 253

N

NewTimer 91
NewTimerSecs 92
NextDisplayInc 653
NextFaxFunc 548
NextFileFunc 319, 320
NoDevice 104
NullModem 255
NumberStr 254

O

OkToCompress (Lzh) 655, 656, 657
OkToCompress (Zip) 693, 694, 695
OkToWrite (Lzh) 655, 656, 657
OkToWrite (Zip) 693, 694, 695
OpenFile 610
otUartPort 159
OutBuffFree 218
OutBuffUsed 218
OutputLineFunc 511

P

PageHeaderRec 510
ParityString 104
ParityType 107
PcxFaxConverter 512
PeekChar 110, 150
PeekCharProc 108
PeekCharTimeout 219
PortRec 108
PortSaveRec 109
PrefixStr 254
PrepareFaxReceivePart (CAS) 611
PrepareFaxReceivePart (C12) 580
PrepareFaxTransmitPart (CAS) 611
PrepareFaxTransmitPart (C12) 580
PrepareReceivePart (non-OOP) 332,
352, 362, 373, 389, 414, 427
PrepareReceivePart (OOP) 332, 352,
363, 374, 390, 415, 428
PrepareTransmitPart (non-OOP) 333,
352, 362, 374, 389, 414, 427
PrepareTransmitPart (OOP) 333, 352,
363, 374, 390, 415, 428
ProcessChar (AnsiEmulator) 464
ProcessChar (TerminalEmulator) 460
ProcessDLE 420
ProcessENQ 421
ProcessESCI 422
ProcessSelf 481
ProtocolData 319
ProtocolInProgress 219
ProtocolRec 319
ProtocolReceive (non-OOP) 334, 352,
362, 374, 389, 414, 427
ProtocolReceive (OOP) 334, 352, 363,
374, 390, 415, 428
ProtocolReceivePart (non-OOP) 335,
352, 362, 374, 389, 414, 427
ProtocolReceivePart (OOP) 335, 352,
363, 374, 390, 415, 428
ProtocolStateType 319
ProtocolTransmit (non-OOP) 336, 352,
362, 374, 390, 414, 427
ProtocolTransmit (OOP) 336, 352,
363, 374, 390, 415, 428
ProtocolTransmitPart (non-OOP) 337,
352, 362, 374, 390, 414, 427
ProtocolTransmitPart (OOP) 337, 352,
363, 374, 390, 415, 428
ProtocolTypeString 318
PS2DetectMode 104
PS2Mode 109
ptErrorProc 160
ptHandleFossilBug 134
ptOptionsAreOn 220
ptOptionsOff 220
ptOptionsOn 220
ptReadWriteWait 138

ptReturnPartialGets 104
 PutBlock 221
 PutBlockDirect 221
 PutBlockTimeout 222
 PutChar 110, 151
 PutCharProc 109
 PutCharTimeout 222
 PutModemCommand 270
 PutString 223
 PutStringTimeout 223

Q

QueueType 595
 QuoteArray 414

R

ReceiveIntMask 105
 RelaxedBlockWait 351
 RelaxedHandshakeWait 351
 RemainingTime 92
 RemainingTimeInMsecs 93
 RemainingTimeInSecs 93
 RemoveModemCode 271
 RemoveModemCommand 271
 RemoveModemResponse 272
 RepeatModemCommand 272
 ResetModem 273
 ResponseType 254
 RestorePort 110, 152
 RestoreUartState 129
 ResWidths 510
 Root 161
 RotatelrqPriority 118
 RunDiagnostics 612

S

SavePort 110, 152
 SavePortProc 109
 SaveUartState 129
 Secs2Tics 94
 SecsPerDay 88
 SendBreak 110, 153
 SendBreakProc 109
 SendLongBreak 130
 SendOutgoingChar 481
 Set1KMode 357
 SetAbortFunc 224
 SetAcceptFaxFunc 554
 SetAcceptFileFunc 338
 SetActualBPS 339
 SetAnswerOnRing 581
 SetArchiveStatus 170
 SetAsyncStatus 170
 SetBackgroundProc 340
 SetBigSubpacketOption 379
 SetBlockWait 357
 SetCaptureFileName 491

SetCarrierTrans 273
 SetCASAabortFunc 613
 SetCASResolution 613
 SetClassType 581
 SetClearScreenProc 452
 SetCompressionMode (Lzh) 669
 SetCompressionMode (Zip) 707
 SetCompressSuccessFunc (Lzh) 670
 SetCompressSuccessFunc (Zip) 708
 SetConnectAttempts 554
 SetConnectState 582
 SetCtlPrefix 396
 SetCurrentAnsiPort 453
 SetDataCompression 274
 SetDataCompression (APCOUR) 293
 SetDataCompression (APMCOM) 294
 SetDCDControl 274
 SetDelays 433
 SetDestinationDirectory 340
 SetDialPrefix (Fax) 582
 SetDialPrefix (Modem) 275
 SetDialPulse 275
 SetDialTime 582
 SetDialTone 276
 SetDTR 225
 SetDTRControl 276
 SetEfficiencyParms 341
 SetEOLChar 433
 SetErrorControl 277
 SetErrorControl (APCOUR) 293
 SetErrorControl (APMCOM) 294
 SetErrorProc 226
 SetExtractSuccessFunc (Lzh) 671
 SetExtractSuccessFunc (Zip) 709
 SetFaxAndData 583
 SetFaxLogProc 555
 SetFaxNameFunc 555
 SetFaxPath 520
 SetFaxPort 584
 SetFaxStatusProc 556
 SetFifoBuffering 130
 SetFileComment 696
 SetFileCommentFunc 710
 SetFileList 342
 SetFileMask 343
 SetFileMgmtOptions 380
 SetFinishWait (XModem) 358
 SetFinishWait (ZModem) 381
 SetFlag 163
 SetFlowControl 278
 SetFlowControl (APCOUR) 293
 SetFlowControl (APMCOM) 294
 SetGMode 358
 SetHandleResponses 279
 SetHandleResumeProc 423
 SetHandshakeWait 343
 SetHeaderText 585
 SetHibitPrefix 396
 SetImplodeFactors 711

SetKermitCheck 397
 SetKermitOptions 398
 SetLine 110, 153
 SetLineProc 109
 SetLinkLocked 280
 SetLinkLocked (APCOUR) 293
 SetLinkLocked (APMCOM) 294
 SetLogFileProc 344
 SetLogoFile 614
 SetMaxLongPacketLen 399
 SetMaxPacketLen 399
 SetMaxRetries 586
 SetMaxTimeoutSecs 400
 SetMaxWindows 400
 SetModem 110, 154
 SetModemCmdMode 281
 SetModemCmdTable 281
 SetModemCodeSet 282
 SetModemCodeTable 283
 SetModemDelay 283
 SetModemEcho 284
 SetModemErrorString 284
 SetModemFeatures 587
 SetModemInit 587
 SetModemOnline 285
 SetModemOnlineEcho 285
 SetModemPort 286
 SetModemProc 109
 SetModemQuiet 286
 SetModemRegister 287
 SetModemRespTable 288
 SetModemResults 289
 SetModemSpeaker 289
 SetModemSpeed 290
 SetModemSpeed (APCOUR) 293
 SetModemSpeed (APMCOM) 294
 SetModemTimeouts 290
 SetModemVolume 291
 SetNextFaxFunc 556
 SetNextFileFunc 345
 SetOkToCompressFunc (Lzh) 672
 SetOkToCompressFunc (Zip) 712
 SetOkToWriteFunc (Lzh) 674
 SetOkToWriteFunc (Zip) 714
 SetOneFax 588
 SetOutputLineFunc 526
 SetOutputPath 643
 SetOverwriteOption 346
 SetPacketPadding 401
 SetPageSize (Fax) 520
 SetPageSize (TERMIN) 482
 SetProgressWidth 675
 SetProtocolPort 346
 SetPS2DetectMode 119
 SetReceiveFilename 347
 SetRecipientName 557
 SetRecoverOption 381
 SetRepeatPrefix 401
 SetResolutionMode 520

SetResolutionWidth 521
 SetRTS 227
 SetSenderName 557
 SetShowCommentsProc 715
 SetShowMethodProc (Lzh) 676
 SetShowMethodProc (Zip) 716
 SetShowNameProc 677
 SetShowProgressFunc (Lzh) 678
 SetShowProgressFunc (Zip) 717
 SetShowStatusProc 347
 SetSoundBellProc 453
 SetSpeedMatching 291
 SetSpeedMatching (APCOUR) 293
 SetSpeedMatching (APMCOM) 294
 SetStationID (Fax converter) 521
 SetStationID (Fax) 558
 SetStatusFunc 522
 SetSWCTurnDelay 402
 SetTag (Lzh) 680
 SetTag (Zip) 717
 SetTaskDate 615
 SetTaskTime 615
 SetTerminalWinEmulator 482
 SetTerminalWinPort 483
 SetTerminator 402
 SetTitle 558
 SetToneDial 588
 SetUart 110, 155, 176
 SetUartProc 110
 SetView 483
 SetWaitCharProc 228
 SetZipComment 718
 sfReceiveFlow 105
 ShowComments (Zip) 693, 694, 696
 ShowMethod (Lzh) 655, 656, 657
 ShowMethod (Zip) 693, 694, 696
 ShowName (Lzh) 655, 656, 657
 ShowProgress (Lzh) 655, 656, 657
 ShowProgress (Zip) 693, 694, 696
 ShowStatusProc 319, 320
 SigArray 510
 SimulateLF 443
 SmallFont 510
 SortFileMaskList 643
 SoundBellProc 451
 SRegSet 254
 Standard 11
 StandardLogging 11
 StandardWidth 510
 StartTracing 119
 StartTransmitter 110, 156
 StartTransmitterProc 110
 Status 11
 StatusArray 595
 StatusBuffering 13, 229
 StatusLogging 11
 StatusRecord 595
 StatusStr 171
 StopBitType 110

- StopTracing 119
- Store (AbstractModem) 292
- Store (AnsiEmulator) 465
- Store (AsciiProtocol) 434
- Store (BPlusProtocol) 424
- Store (CaptureTerminalWindow) 492
- Store (Digi14Port) 230
- Store (FossilPort) 230
- Store (Int14Port) 230
- Store (KermitProtocol) 403
- Store (TerminalEmulator) 460
- Store (TerminalWindow) 484
- Store (UartPort) 230
- Store (XModemProtocol) 359
- Store (ZmodemProtocol) 382
- SubmitSingleFile 616
- SubmitTask 617
- SupportsBatch 348
- SWCKermitTurnDelay 389
- SWFlowCtl 110, 156
- SWFlowDisable 232
- SWFlowEnable 233
- SWFlowEnableOpt 234
- SWFlowGet 110, 157
- SWFlowResume 235
- SWFlowSet 110, 157
- SWFlowSetChars 236
- SWFlowState 236
- SWOnly 11

T

- TelixaDelay 318
- teMapVT100 458
- teOptionsAreOn 461
- teOptionsOff 461
- teOptionsOn 461
- TerminalCommands 475
- TerminalEmulator 458
- TerminalKeySet 473
- TerminalWindow 475
- TextFaxConverter 512
- Tics2Secs 94
- TicsPerDay 88
- TiffFaxConverter 512
- TimerExpired 94
- TLogFaxCode 547
- Tracing 12
- TracingOn 105
- TransReady 110, 158
- TransReadyFunc 110
- twFloatingHView 473
- twOptionsAreOn 485
- twOptionsOff 485
- twOptionsOn 485

U

- UartPort 187
- UartTest1 131

- UartTest2 132
- UartTest3 132
- UartType 125
- UartTypeString 125
- ucTermWin 474
- ufHalfWidth 510
- ufOptionsAreOn 527
- ufOptionsOff 527
- ufOptionsOn 527
- UnLzh 657
- UnpackFax 511, 512
- UnpackFile 528
- UnpackFileToPcx 528
- UnpackPage 529
- UnpackToPcx 511, 512
- UnZip 695
- UpdateChecksum 164
- UpdateCommentsFileMaskList 718
- UpdateCrc 164
- UpdateCrc32 165
- UpdateCrcKermit 165
- UpdateInterval 510
- UpdateLineStatus 110, 158
- UpdateLineStatusFunc 110
- UpdateModemStatus 110, 158
- UpdateModemStatusFunc 110
- UseHWFlow 13
- UseOOP 12
- UseOPro 12
- UsePmodeDLL 12
- UserBackProc 319, 320
- UserDefined 11
- UseSWFlow 13
- UseTPro 12
- UseVT100Mode 451

V

- ValidLineStatus 135
- ValidModemStatus 135
- veUartPort 160

W

- WaitCharProc 186, 187
- WaitForChar 237
- WaitForMultiString 237
- WaitForString 238
- WindowsUsed 405
- WordLenOMask 105
- WriteChar 486
- WriteCharAnsi 454
- WriteCharAnsiPort 454
- WriteFailOptions 318
- WriteNewerLonger 372
- WriteString 486
- WriteStringAnsi 455
- WriteStringAnsiPort 455

X

XmodemOverhead 351
XmodemProtocol 351, 352
XmodemTurnDelay 351

Y

YmodemProtocol 362, 363

Z

Zip 695
ZipComment 692
ZipFileList 693, 695
ZipNode 693, 695
ZmodemOverHead 373
ZmodemProtocol 373, 374
ZmodemTurnDelay 373
ZrQinit 373

Index

0

16-bit CRC 164
32-bit CRC 165

A

Abort function 224
Abort hook 301
Abstract Layer 76, 95, 733
AbstractModem 81
AbstractPort 80, 175
AbstractProtocol 81
Accept fax hook 545
Accept file function 306, 338
Add-on layer 79, 733
ANSI 733
ANSI escape sequences 448, 454, 455, 456, 458, 467
ANSI.SYS 81, 447, 454
AnsiEmulator 81
APDEFINE.INC 10, 14, 59
APF file format 503
APF files 507
Appendices 15
APRO.MAK 6
APRO.TXT 52
Archive 631
 create output file 638
 error 640
 get name 640
 options 637, 638
 output path 643
Archive object
 dispose 639
 initialize 641
ArchiveStatus 167
ARCNET network board 18
ASCII protocol 426
 delay 433
 EOL 433
 line number 429
ASCII protocol object
 dispose 429
 initialize 430, 431
 load 432
 store 434
ASCII protocol record
 dispose 429
 initialize 430, 431
ASCII text documents 506
AsciiProtocol 81
Asynchronous serial communication 64, 733
AsyncStatus 167
AT Commands 242, 733
Automatic response handling 279

B

BACK/BACKO 438
Background fax transfer 533
Background file transfer 312
Background tasks 471
Background user hook 340
Batch file transfers 348
Baud rate
 change 188
 definition of 65, 733
Bell 103 733
Bell 212A 733
Bit flags
 clear 162
 set 163
 test 162
Bit rate 65
Block number 324
Block size 324
BPlus protocol
 flow 406
 interrupted transfer 412
 name collision 423
 process DLE 420
 process ENQ 421
 process ESCI 422
 quoted data 407
 terminal processing 408
BPlus protocol object
 dispose 416
 initialize 416, 417
 load 419
 store 424
BPlus protocol record
 dispose 416
 initialize 417, 418
BPS 733
Break 67, 733
 check 196
 long 130
 send 153
 signal 153
Buffers
 change limits 202
 change size 189
 delay 199
 flush 143, 201
 free space 213, 218
 get character location 206
 space used 213, 218
 usage information 144
Bytes remaining to be transferred 325

C

Capture buffer 470, 489
Capture file 470, 489

- Capture terminal window
 - check on 487
 - dispose 487
 - file name 488, 491
 - initialize 488, 489
 - load 490
 - store 492
 - turn on/off 487
 - CaptureTerminalWindow 81
 - CAS Fax 733
 - abort 596
 - abort function 613
 - auto-receive 608
 - close file 596
 - date 605, 615
 - delete a file 597
 - delete files 597
 - diagnostics 612
 - driver info 606
 - error codes 593
 - find first event 603
 - find next event 603
 - hardware status 607
 - installed 596
 - move file 610
 - open file 610
 - page info 607
 - queue status 608
 - queues 589
 - receive 590, 599, 600, 611
 - resolution 613
 - status 604, 605
 - status hook 592
 - submit task 617
 - time 606, 615
 - transmit 590, 601, 602, 611, 616
 - user abort hook 592
 - CAS fax object
 - dispose 598
 - initialize 609
 - CAS fax record
 - dispose 598
 - initialize 609
 - CCITT 733
 - Central directory of a ZIP file 690
 - Character quoting in Kermit 384
 - Character-time 734
 - Checksum 734
 - calculate 164, 165
 - Class 1/2 498, 734
 - Clear to send 74, 184
 - Clock ticks 87
 - Columbia University 384
 - Com port 64, 734
 - get name string 113
 - Command and response tables 243
 - CommandProcessor 81
 - CommandWindow 81
 - Compiler Options 13
 - CompuServe 59, 62, 65, 357, 413
 - B+ protocol 406
 - COMTEST/COMTESTO 18
 - Conditional defines 10
 - Connect speed 263
 - Control Character Escaping, Zmodem 369
 - Conversion Status Hook 508
 - Core routines 76, 95, 96, 142, 175, 734
 - Core virtual methods 175
 - CourierModem 81
 - Cover page for a fax 535
 - CRC 734
 - CTS 74, 183, 184, 734
 - check 191
 - Current number of block errors 323
 - CVT2FAX/CVT2FAXO 621
- ## D
- Data Bits 65, 734
 - change 189
 - Data carrier detect 74, 192, 274
 - Data compression 384, 734
 - in modems 274
 - Data ready 191
 - Data set ready 74, 184, 194
 - Data Terminal Ready 73, 147, 154, 184, 225
 - DCD 74, 734
 - check 192
 - set behavior 274
 - DCE 734
 - Debugger 743
 - Delay calibration 89
 - Delimiters 173
 - Delta CTS 74
 - check 192
 - Delta DCD 74
 - check 193
 - Delta DSR 74
 - check 193
 - Delta RI 74
 - check 194
 - Demonstration programs 7
 - DesqView 89
 - Device 64
 - Device independence 76
 - Device layer 77, 95, 734
 - Dial prefix 261, 275
 - DialModem 261
 - Digi14
 - activate 139
 - DigiBoard Universal Device Driver 137
 - Divisor latch high 70
 - Divisor latch low 69
 - DMA access 71
 - Document conversion 497

DOS Exec function 143, 145

DR

check 191

DSR 74, 184, 734

check 194

DTE 734

DTR 73, 147, 154, 184, 734

set 225

set behavior 276

Dumb terminal 447, 737

E

EMS memory 53

Enquire Control Sequence 408

Error

codes 100, 166, 739

correcting connection 264

correction 277, 735

handler 59, 99, 147, 226, 476

handling 166, 301, 634

message 171

number 166

report to user error handler 147

Escape sequences 448, 737

EvenParity 66

Event logging

dump report 127

initialize 128

EventLogging 59, 99, 120, 744

EventTimers 87

Extracting files from a ZIP archive 684, 730

Extracting files from an LZH archive 647, 723

F

FASTUPD.xxx 3

Fax 735

accept fax function 554

add entry 549

and data 583

APF file format 503

APF files 507

ASCII text documents 506

background transfer 533

Class 1/2 498

class type 581

clear entries 551

com port 584

connect retry 554

connect state 582

dial prefix 582

dial timeout 582

document conversion 497

from a TSR 534, 535

group 1,2,3 498

hangup 569

header 585

in progress 200

initialize modem 579

logging 540

logging procedure 555

modem class 572

modem features 573, 587

modem info 574

modem initialization 587

name 552

naming function 555

next fax function 556

one fax feature 588

options 550

page info 575

page status 571

PCX graphic images 507

receive 559, 565, 566, 580

recipient name 557

retries 586

rings to answer 581

sender name 557

session parameters 576

station ID 558, 575

status 539

status procedure 556

TIFF graphic images 507

title 558

tone or pulse dial 588

transmit 559, 567, 568, 580

Fax convert

convert file 513

dispose 514

font 518, 519

get file name 516

initialize 517

options 515

page size 520

resolution 520, 521

set path 520

station ID 521

status 516

status function 522

Fax object

dispose 551, 563

initialize 553, 577

Fax record

dispose 564

initialize 578

Fax unpack

dispose 523

header 524

initialize 525

options 527

output line function 526

page header 524

unpack file 528

unpack page 529

FAX2PCX/FAX2PCXO 625

Faxmodem 735

FAXSRVR/FAXSRVRO 628

FIFO

- buffering 122, 127
- buffering on/off 130
- control register 70
- error 195
- mode 71, 122, 735
- status 127

File list

- add name 321
- allocate 331
- dispose 322
- set 342

File mask list 632

- add file mask 637
- deallocate 639
- expand 640
- initialize 641
- match file name 642, 643
- set mask 343
- sort 643

File mask node

- deallocate 639
- initialize 641
- match file name 642

File recovery option, Zmodem 381

File transfer protocols 295

FileMode 641, 668, 706

Floating view 467, 735

Flow Control 181, 735

Fossil

- activate 136
- device driver 133

Frame, Zmodem 368

Framing errors 67

Full duplex 735

Full-screen zoom 471

FX/FXO 436

G

Greenwich Meridian Time 318

Group 1,2,3 498, 735

H

Half duplex 735

Handshaking 73, 182, 735

Hardware flow control 73, 154, 182

disable 210

enable 211

get state 148, 212

options 183

start/stop 148

Hardware interrupts 75

Hayes SmartModem 242

HDLC 735

Help file 52

I

IBM PC standard 68

IBM PS/2 standard 68

Int14

activate 141

Int14Port 81

Interface layer 78, 96, 173, 735

Interrupt enable register 70

Interrupt identification register 70

Interrupted transfer 412

Interrupts 75

activate for port 143

IRQ 75, 735

rotate priority 118

K

Kermit 384

block check 397

CRC 165

high bit quote prefix 396

long packets 387, 392, 399

options 385, 398

packet length 399

packet terminator 402

packets 385

padding character 401

quote character 396

repeat quote prefix character 401

server 387

Sliding Windows Control 387

SWC 400

SWC slots available 392

SWC turnaround delay 402

wait time 400

windows used 405

Kermit object

dispose 391

initialize 393

load 395

store 403

Kermit protocol record

dispose 391

initialize 394

Kernel 120, 735

L

LAP M 736

Layered architecture 76

Lead-in Sequence 408

LHA 631

LHA.EXE 3

LHARC 631

Line control byte 206

Line control register 72

Line error 65, 66, 185, 736

check 196

get 207

Line feed 433
Line parameters 64, 173
Line status 173
Line status register 73
 get 207
Link speed between two modems 290
Log file procedure 344
Logging procedure for protocols 303
Long packets 387

LZH

 compress 659
 compress success function 670
 create archive 660, 661
 dispose 664
 extract file 665, 666
 extract success function 671
 file 644
 file structure 653
 initialize 668
 initialize archive 667
 initialize node 667
 ok to compress function 672
 ok to write function 674
 open archive file 668
 progress display width 675
 set compression mode 669
 show method procedure 676
 show name procedure 677
 show progress function 678
 tag file 680
 update files 666
 updating 649

LZH file list

 add file 658
 build 658
 delete file 661, 662, 663
 dispose 663, 664
 dispose node 663
 initialize 667, 669

LZH/LZHO 719

LZHV/LZHVO 721

LZHX/LZHXO 723

M

MAKEHELP 52

Managed modem link 249, 274, 277,
 278, 280, 290, 291, 736

MarkParity 66

Memory

 allocate 163
 deallocate 163

Microcom 736

Microcom Network Protocol 249

Microcom QX/3296c modem 241, 294

MicrocomModem 81

MNP 249, 264, 736

Modem 239, 736

 answer immediately 259

 auto answer 260
 automatic response handling 279
 command mode 281
 command result 266
 configuration 247
 control register 73, 208
 data compression 274
 data rate 290
 dial 261
 echo 284, 285
 error control 277
 error string 284
 errors 248
 flow control 248
 hang up 267
 library 247
 lock connection rate 280
 pulse dialing 275
 quiet 286
 repeat last command 272
 reset 273
 send command 263, 270
 set BPS 339
 set carrier 273
 set delay 283
 set dial prefix 275
 set flow control 278
 set port 286
 speaker 289
 speed matching 291
 status register 74, 208
 terminal mode 285
 timeouts 290
 tone dialing 276
 transfer time 341

Modem command

 add 257
 remove 271
 set table 281

Modem object

 dispose of 262
 initialize 267
 load 269, 292

Modem record

 dispose of 262
 initialize 268

Modem response

 add 258
 add numeric 256
 get code 264
 get text 265
 remove 272
 remove numeric 271
 set numeric table 283
 set table 282, 288
 speaker volume 291
 words or codes 289

MODLIB.LZH 247

Modules provided 6

Multiple port boards 76
Multiple ports 69
Multitasking a protocol 315

N

Network shared-modem pool 76, 102, 736
Next fax hook 542
Next file function 304, 345
NoParity 66
NullModem 81

O

Object Professional 2, 12, 22, 95, 159, 456, 466
OddParity 66
On-screen clock 471
OOPCOM 25
OOPINST 37
Organization of this manual 15
OutputLine hook 508
Overlaid, APRO units that can be 13
Overrun error 124

P

PACKING.LST 3
Page size 482
Parallel communications 64
Parity 736
 bit 65
 change 190
 errors 67, 745
PCX graphic images 507
Pixel 736
PKZIP 631
POPHELP 52
Port 736
 character ready 144
 check ready 158
 close 145
 deactivate 145
 get character 146
 get line parameters 146
 get line status register 158
 get modem signals 147
 get modem status register 158
 get name 205
 initialize 214
 initialize line 153
 open 149, 150
 options 220
 peek character 150
 put character 151
 restart transmission 156
 restore state 152
 save state 152
 set modem signals 154

Port object
 load 215
 store 230
PortRec 95, 96, 102
PRINTDOC.EXE 3
PRNFAX/PRNFAXO 624
ProComm 388
PROTDOC.LZH 361, 368
Protocol 295, 736
 aborting 301
 batch file transfer 348
 block number 324
 block size 324
 bytes remaining 325
 bytes transferred 325
 CRC type 326
 current block number 326
 current file 327
 current file pathname 329
 current file size 328
 current protocol type 330
 elapsed time 327
 errors 323
 handshake 343
 in progress 219
 initial file position 328
 number of errors 330
 options 321, 322
 overwrite file 346
 receive 334
 receive, background 332, 335
 receive file name 347
 set port 346
 status 347
 transfer time 323
 transmit 336
 transmit, background 333, 337
Protocol object
 dispose 322
 initialize 331
ProtocolInProgress 100, 219
PS/2 120
 check for 117
 set detect mode 119
Pulse (rotary) dialing mode 275
Purchase Agreement 61

Q

Quick B 406
Quoted data 407

R

Raster line 736
READ.1ST 3, 16
READ.ME 3
ReadLn 442

- Receive
 - block 203, 204
 - character 205
 - check for string 112
 - peek character 219
 - ready 188
 - set directory 340
 - string 209, 210
 - wait for character 237
 - wait for multi string 237
 - wait for string 238
- Receiver buffer register 69
- Receiving files with a protocol 307
- Reference sections of the manual 15
- Registers 69
- Relaxed Xmodem 351, 357
- Remote device 64, 736
- Request To Send 73, 147, 154, 184, 227
- Rerouting the cable 67
- RFAX/RFAXO 626
- RI 74, 736
 - check 197
- Ring indicator 74, 197
- Root object 80, 161
- RS-232 182, 736
- RTS 73, 147, 154, 183, 184, 737
 - set 227
- Run length encoding 384
- S**
 - S-registers 254, 737
 - get 265
 - set 287
 - Scroll back buffer 466, 479
 - Self-extracting archive 727, 730
 - Self-extracting files 644, 681
 - Serial communications 64
 - Serial data 737
 - Serial ports 64
 - SHOWFAX/SHOWFAXO 622
 - SIMPCOM 22
 - SIMPRCV/SIMPRCVO 620
 - SIMPSND/SIMPSNDO 618
 - Sliding Windows Control 387
 - Smart terminal 737
 - Software flow control 182, 184, 369
 - disable 232
 - enable 233, 234
 - get state 157
 - resume 235
 - set characters 156, 236
 - start/stop 157
 - state 236
 - Software interrupts 75
 - SpaceParity 66
 - Speaker
 - modem 289
 - volume 291
 - Standalone 159
 - Start bit 737
 - Status
 - get 169
 - set 170
 - Status Buffering 185
 - disable 198
 - enable 200
 - state 229
 - Status procedure for protocols 301
 - Stop Bits 65, 737
 - change 190
 - Stream I/O 160
 - Streaming protocol 351, 737
 - Streams 161
 - Subpacket, Zmodem 368
 - Support
 - CompuServe 62
 - telephone 62
 - Synchronous communications 64
- T**
 - Tag 686
 - Tagging 649
 - Technical support 59
 - Telephone support 62
 - Terminal 447, 737
 - Terminal emulation (non-OOP)
 - clear screen 452
 - port 453
 - sound bell 453
 - write character 454
 - write string 455
 - Terminal emulation (OOP)
 - dispose 459, 462
 - initialize 459, 462
 - load 459, 463
 - options 461
 - process character 460, 464
 - store 460, 465
 - Terminal emulator 737
 - Terminal window
 - capturing a session 470
 - change object 482, 483
 - commands 472
 - dispose 476
 - flow control 472
 - get character 477
 - initialize 478, 479
 - load 480
 - object pointer 477
 - options 485
 - port object pointer 478
 - process 481
 - store 484
 - transmit character 481
 - viewport 483

- Terminal window
 - virtual screen 476, 482
 - with BPlus 410
 - write character 486
 - write string 486
 - TerminalEmulator 81
 - TerminalWindow 81
 - TERMTEST 494
 - Text file device driver 441
 - assign port 444
 - get port ptr 445
 - Throughput 65
 - TIA/EIA 737
 - TIFF graphic images 507
 - Time
 - elapsed 90, 91
 - Timer
 - convert seconds to ticks 94
 - convert ticks to seconds 94
 - delay 89
 - functions 87
 - remaining time 92, 93
 - start 91, 92
 - test expired 94
 - Tone dialing 276
 - TPFAX 43
 - Trace buffer 115
 - Tracing 59, 97, 120, 744
 - abort 111
 - add entry 111
 - clear buffer 113
 - dump buffer 114
 - dump buffer in hex 114
 - initialize queue 115
 - start 119
 - stop 119
 - Trailing edge ring indicator 74, 194
 - Transmit
 - block 221, 222
 - character 222
 - string 223
 - Transmitter empty 197
 - Transmitter holding register 69, 73, 158, 197
 - Transmitter shift register 73
 - Transmitting files with a protocol 309
 - Trigger level 123
 - TSR, sending a fax from 534
 - TSR, using protocols in 315
 - Turbo Professional 2, 95, 159
 - Turbo Vision 456
- U**
- U.S. Robotics Courier modem 241, 293
 - UART 64, 737
 - activate 126
 - get base address 202
 - overrun 66, 744
 - registers 69
 - restore state 129
 - save state 129
 - set 155
 - test 131, 132
 - types 126
 - UARTID 17
 - UartPort 81
 - Unbuffered device layer 140
 - Universal Asynchronous
 - Receiver/Transmitter 64, 68
 - UseOPro 95, 159, 470
 - User abort function (port) 101
 - User abort hook (Fax) 537
 - User background hook (Protocol) 313
 - User error hook (Fax) 537
 - User error procedure (Port) 99
 - UseTPro 95, 159
- V**
- V.17 737
 - V.21 737
 - V.22 737
 - V.25bis 738
 - V.27 738
 - V.29 738
 - V.32 738
 - V.42 264, 738
 - V.42 LAPM 249
 - Viewing the Directory of a ZIP Archive 681
 - Viewport 466, 738
 - Virtual cursor 466
 - Virtual screen 466, 476
 - VT100 447, 458, 737
- W**
- Wait vs. no-wait I/O 180
 - WaitChar procedure 180, 228
 - WaitFor procedures 180
 - WriteLn 441
- X**
- Xmodem 349
 - 1K 350, 357
 - 1KG 350, 358
 - block wait 357
 - CRC 350
 - finish wait 358
 - Xmodem 1K 354
 - Xmodem 1KG 354
 - Xmodem CRC 354
 - Xmodem extensions 350
 - XModem object
 - initialize 353, 354
 - load 356
 - store 359

- XModem protocol record
 - dispose 353
 - initialize 355
- XmodemProtocol 81
- XMS memory 54
- Xoff 184, 735
- Xon 184, 735

Y

- Ymodem 361
 - extensions 362
 - G 362
- Ymodem G 365
- YModem object
 - dispose 364
 - initialize 365
 - load 367
- YModem protocol record
 - dispose 364
 - initialize 366
- YmodemProtocol 81

Z

- ZIP
 - archives 681
 - close archive 702
 - comment 718
 - compress 697, 698
 - compress success function 708
 - compression mode 707
 - create archive 698, 699
 - delete file 700
 - destroy node 701
 - extract file 702, 703
 - extract success function 709
 - file comment function 710
 - file structure 690
 - implode factors 711
 - initialize 704
 - initialize archive 705
 - initialize node 705
 - ok to compress function 712
 - ok to write function 714
 - open archive 706
 - show comments procedure 715
 - show method procedure 716
 - show progress function 717
 - tag file 717
 - update comments 718
 - update files 704
 - updating 686
 - viewing directory 721, 727
- ZIP file list
 - add file 697
 - build 697
 - delete node 699, 701
 - destroy 701, 702

- initialize 705, 707
- ZIP/ZIP0 725
- ZIP2EXE 681, 727, 730
- ZIPV/ZIPVO 727
- ZIPX/ZIPX0 730
- Zmodem 368
 - 8K option 379
 - block sizes 371
 - file management options 380
 - finish wait 381
 - large block support 371
 - protocol options 369
- ZModem object
 - dispose 375
 - initialize 376
 - load 378
 - store 382
- ZModem protocol record
 - dispose 375
 - initialize 377

Async Professional 2.0TM

Powerful Serial Communications Tools for Dos Applications

Async Professional is a full-featured communications toolkit that enables you to write async applications faster and easier. It provides powerful features like:

- OOP and procedural calling interfaces
- Interrupt driven I/O to 115 Kbaud
- XON/XOFF, CTS/RTS, and DTR/DSR flow control
- UART, DigiBoard, Fossil, and BIOS com ports
- 16550 buffered UART support
- ZMODEM, YMODEM, XMODEM, CIS B+, and Kermit protocols
- Modem control including Hayes, V.32, V.42, and MNP5
- Trace and event logging facilities for com debugging
- ANSI terminal emulation
- Class I, Class II, and CAS fax support
- ZIP and LZH data compression and decompression

Hot Demo Programs Accelerate your Learning

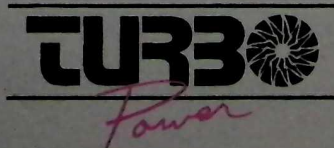
Async Professional includes examples ranging from simple ones to get you started to sophisticated applications that you can use daily. The demos include:

- A general purpose multi-window com program with menus, text editor, phone book, upload/download support, and more, built using Object Professional.
- A diagnostic com program that shows incoming and outgoing characters, UART signals, and more.
- A fax control center that allows you to convert, send, view, and print faxes.
- TSRs for performing file transfer protocols and fax receives in the background.
- Compression demos that provide many of the features of PKZIP and LHA.

Full Source Code, Expert Support, and No Royalties

Async Professional includes complete source code, comprehensive documentation, pop-up help, and free technical support by telephone, voice, and fax. You pay no royalties for applications written using Async Professional.

Async Professional requires Turbo Pascal 6.0 or 7.0, or Borland Pascal 7.0. MS-DOS 2.0 or later, and an IBM AT, PS/2, or compatible are also required. Dual media included. Async Professional is compatible with Object Professional, Turbo Professional, Turbo Vision, and other DOS user interface libraries.



TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009
CompuServe I.D. 76004, 2611

©TurboPower Software 1993